SOLVING BIG DATA PROBLEMS
from Sequences to Tables and Graphs

FELIX HALIM

Bachelor of Computing
BINUS University

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2012

Acknowledgements

First and foremost, I would like to thank my supervisor Prof. Roland Yap for introducing and guiding me to research. He is very friendly, supportive, very meticulous and thorough in reviewing my research. He gave a lot of constructive feedbacks even when the research topic was not in his main areas.

I am glad I met Dr. Panagiotis Karras in several of his lectures on the Advanced Algorithm class and Advanced Topics in Database Management Systems class. Since then we have been collaborating in advancing the state of the art of the sequence segmentation algorithms. Through him, I get introduced to Dr. Stratos Idreos from Centrum Wiskunde Informatica (CWI) who then offered an unforgettable internship experience at CWI which further expand my research experience.

I would like to thank to all my co-authors in my research papers: Yongzheng Wu, Goetz Graefe, Harumi Kuno, Stefan Manegold, Steven Halim, Rajiv Ramnath, Sufatrio, and Suhendry Effendy. As well as the members of the thesis committee who have reviewed this thesis: Prof. Tan Kian Lee, Prof. Chan Chee Yong, and Prof. Stephane Bressan.

Last but not least, I would like to thank my parents, Tjoe Tjie Fong and Tan Hoey Lan, who play very important role in my development into a person I am today.

Contents

Summary

Big Data problems arise when the existing solutions become impractical to run because the amount of resources needed to process the ever increasing amount of data exceeds the available resources which depend on the context of each application. Classical problems whose solutions consume resources with more than linear in complexity will face the big data problem sooner. Thus, such problems that were considered *solved* need to be revisited in the context of big data. This thesis provides solutions to three big data problems and summarizes the shared important lessons such as stochasticity, robustness, inherent properties of the underlying data, and algorithm-system optimizations.

The first big data problem is the sequence segmentation problem also known as histogram construction. It is a classic problem on summarizing a large data sequence to a much smaller (approximated) data sequence. With limited amount of resources available, the practical challenge is to construct a segmentation with as low error as possible and consumes as few resources as possible. This requires the algorithms to provide good tradeoffs between the amounts of resources spent versus the result quality. We proposed a novel stochastic local search algorithm that effectively captures the characteristics of the data sequence and quickly discovers good segmentation positions. The stochasticity makes it robust to be used for generating sample solutions that can be recombined into a segmentation with significantly better quality while maintaining linear time complexity. Our state-of-the-art segmentation algorithms scale well and provide the best tradeoffs in terms of quality and efficiency, allowing faster segmentation for larger data sequences than existing algorithms.

In the second big data problem, we revisit the recent work on adaptive indexing. Traditional DBMS has been struggling in processing large scientific data. One major bottleneck is the large initialization cost, that is to process queries efficiently, the traditional DBMS requires both knowledge about the workload and sufficient idle time to prepare the physical data store. A recent approach, *Database Cracking* [53], alleviates this problem via a form of incremental-adaptive indexing. It requires little or no initialization cost (i.e, no workload knowledge or idle time required) as it uses the user queries as advice to refine incrementally its physical datastore (indexes). Thus cracking is designed to quickly adapt to the user query workload. Database cracking has the philosophy of doing *just enough*. That is, only process data that are directly relevant to the query at hand. This thesis revisits this philosophy and shows that it can backfire as being fully driven by the user queries may not be ideal in an unpredictable and dynamic environment. We show that this cracking philosophy has a weakness, namely

that it is not robust under dynamic query workloads. It can end up consuming significantly more resources that it should and even worse, it fails to adapt (according to cracking philosophy). We propose *stochastic cracking* that relaxes the philosophy to invest some small computation that makes it an overall robust solution under dynamic environment while maintaining the efficiency, adaptivity, design principles, and interface of the original cracking. Under a real workload, stochastic cracking answered the $1.6 * 10^5$ queries up to two orders of magnitude faster compared to the original cracking while the full indexing approach is not even halfway towards preparing a traditional full index.

Lastly, we revisit the traditional graph problems whose solutions have quadratic (or more) runtime complexity. Such solutions are impractical when faced with graphs from the Internet due to the large graph size that the quadratic amount of computation needed simply far outpaces the linear increase of the compute resources. Nevertheless, most large real-world graphs have been observed to exhibit small-world network properties. This thesis demonstrates how to take advantage the inherent property of such graph, in particular, the small diameter property and its robustness against edge removals, to redesign a quadratic graph algorithm (for general graphs) into a practical algorithm designed for large small-world graphs. We show empirically that the algorithm provides a linear runtime complexity in terms of the graph size and the diameter of the graph. We designed our algorithms to be highly parallel and distributed which allows it to scale to very large graphs. We implemented our algorithms on top of a well-known and well-established distributed computation framework, the MapReduce framework, and show that it scales horizontally very well. Moreover, we show how to leverage the vast amount of parallel computation provided by the framework, identify the bottlenecks and provide algorithm-system optimizations around it.

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Big Data Problems

We are in the era of *Big Data*. Enormous amount of data are being collected everyday in business transactions, mobile sensors, social interactions, bioinformatics, astronomy, etc. Being able to process big data can bring significant advantage in making informed decisions, getting new insights, and better understanding the nature. Processing big data starts to become problematic when the amount of resources needed to process the data grow larger than the available computing resources (as illustrated in Figure 1.1). The resources here may represent a combination of available processing time, number of CPUs, memory/storage capacity, etc.

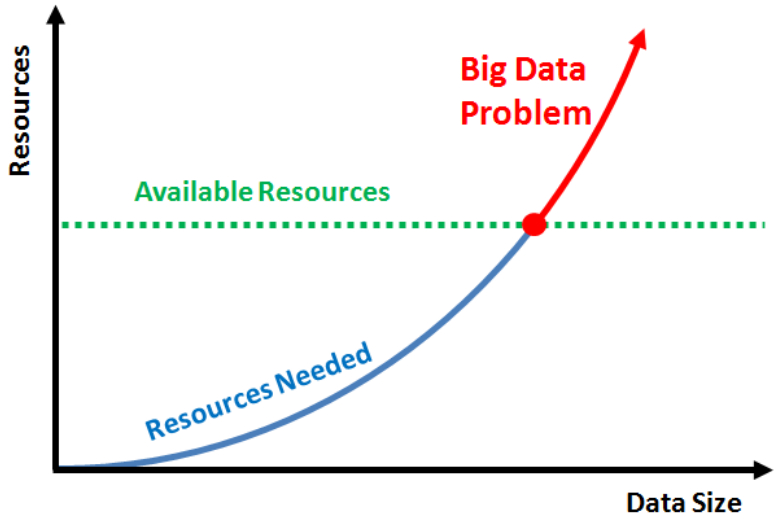


Figure 1.1: Big Data Problem

There could be many different solutions (i.e., techniques or algorithms) to solve a given a problem. Different solutions may require different amount of

resources. Moreover, different solutions may give different tradeoffs in terms of amount of resources needed versus quality of result produced. Considering the limited resources available and rapidly increasing data size, one must carefully evaluate the existing solutions and pick the one that works within the resources capacity and provides acceptable result quality in order to scale to large data sizes. What we considered as big data problems are *relative* to the amount of available resources which depends on the type of applications and contexts where the solutions are applied. That is, on applications with abundance amount of resources, the solutions may work perfectly fine, however the solutions may face big data problems under environments with very limited resources. Typically, solutions that consume *more than linear* amount of resources in proportion to the data size will run into the big data problem sooner. Understanding the tradeoffs of the existing solutions may not be enough because one may require an entirely new solution as the existing (traditional) solutions becomes too ineffective/inefficient.

It is the role of *Data Scientist* to deal with these complex analyses and come up with a solution in solving big data problems. A recent study showed that data scientist is on high demand for the next 5 years and has outpaced the supply of talent [3]. In this thesis, we will play the role of a data scientist and evaluate existing solutions of the three kinds of big data problems, propose new and/or improve on existing solutions, then summarize the important lessons learned.

This chapter gives a brief overview of the three big data problems. These problems exists in different scales in terms of number of available resources and data size as illustrated in Figure 1.2. In the limited scale (a), such as sensor networks, we have the sequence segmentation (or histogram construction) problem. In desktop/server scale (b), we have database indexing problem. In cloud computing scale (c), we have large graph processing problem. The solutions to these seemingly unrelated big data problems share many common aspects, namely:

- *(Sub)Linear in Complexity.* The (sub)linear complexity is the ingredient for scalable algorithms. We designed new algorithms for (a) and (c) that reduce the complexity to linear and relaxed the algorithm for (b) to give robust sub-linear complexity.

- *Stochastic Behavior.* Stochasticity (and/or non-determinism) is used for (a) and (b) to bring robustness into the algorithms and for (c) to be more efficient in queue processing.

- *Robust Behavior.* Algorithm robustness is paramount as without it any algorithm will fail to achieve whatever goals it set out to achieve.

- *Effective exploitation of inherent properties of the data.* By exploiting the

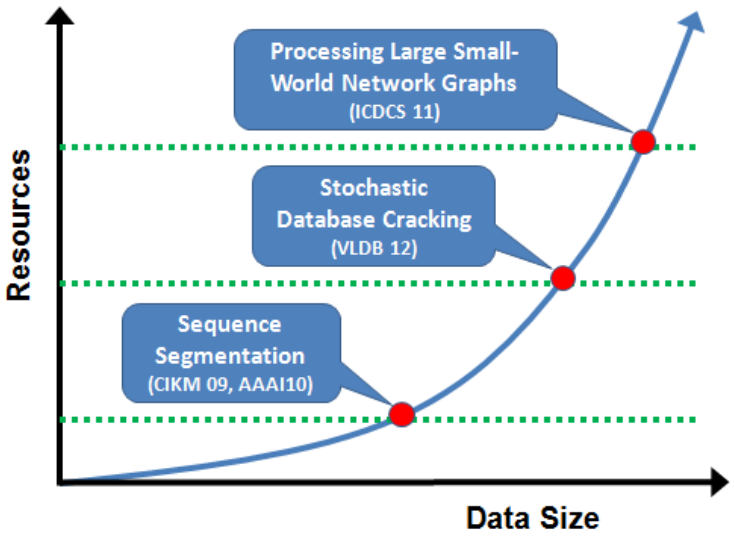


Figure 1.2: The different scales of the three big data problems

inherent properties of the data, a significantly more efficient algorithm can be designed for (c). The characteristics of data can also be used to improve the sampling effectiveness for (a) and to be used as trigger stochastic action in (b).

1.1.1 Sequence Segmentation

The sequence segmentation is the problem of segmenting a large data sequence into a (much smaller) number of segments. Depending on the context, the *sequence segmentation problem* can be seen as *histogram construction* problem or a problem of creating a synopsis of a large data sequence into a much smaller (approximated) data sequence. Sequence segmentation problems arise in many application areas such as mobile devices, database systems, telecommunications, bioinformatics, medical, financial data, scientific measurements, and in information retrieval.

With ever increasing size of the data sequence, sequence segmentation becomes a big data problem in many application settings. Imagine a mobile device that requires context awareness capability [52]. Context awareness can be inferred by analyzing the signals captured by different sensors. These sensors often produce large time series data sequence that need to be summarized to a much smaller sequence (which can be seen as a sequence segmentation problem). However, mobile devices have limited amount of resources to process data produced by the sensors (i.e., limited battery life and computing power). The optimal sequence segmentation algorithm quickly becomes impractical due to its quadratic run-

time complexity. The existing heuristics are shown (in this thesis) to have poor segmentation quality. Recent research has revisited the segmentation problem in the point of view of approximation algorithms. However, it still impractical for large data sequence and failed to resolve the tradeoffs between efficiency versus quality.

In this thesis, a novel state-of-the-art sequence segmentation algorithm is proposed which matches or exceeds the quality of existing approximation algorithms while having performance of existing heuristics. Moreover, we provide extensive comparisons to the existing various sequence segmentation algorithms measured on its quality and efficiency on various well known datasets. The proposed algorithm has linear runtime complexity on the size of the data sequence and on the number of segments generated. The algorithm works by combining the strength of stochastic local search in consistently generating good samples and the existing optimal algorithm to recombine them into a final segmentation with significantly better quality. Our local search algorithm is targeted towards finding good segmentation positions that are relevant to the data. This technique turns out to be far more effective than the approximation algorithms which are targeted towards lowering the total error. We show that in practice, the algorithm practically produces high-quality segmentation on very large data sequences where existing approximations are impractical and existing heuristics are ineffective.

1.1.2 Robust Cracking

Scientific data tends to be very large both in terms of the number of tuples and its attributes. For example, a table in the SkyServer dataset has 466 columns and 270 million rows [64]. New datasets may arrive periodically and the queries imposed on scientific data are very dynamic (i.e., it do not necessarily follow a predetermined pattern) and unpredictable (i.e., it may depend on the previous query result, or it can be arbitrary/exploratory). These characteristics pose as an interesting challenge in creating efficient query processing system.

Traditional database management systems rely heavily on indexing to speedup the query performance. However, existing indexing approaches such as offline and online indexing fail under dynamic query workloads. Offline indexing works by first preparing the physical data store for efficient access. The preparation requires knowledge of the query workload beforehand which is scarce in dynamic environment. Normally, the preparation is tantamount to fully sorting the data so that queries can be answered efficiently using binary search. This preparation costs becomes the biggest bottleneck if the number of elements in the data is extremely large. Moreover, the preparation costs may be overkill if the data is

only queried for a few times before the user move on to the next dataset. That is, it may be better to perform linear scans if there are only a dozen or so queries.

Most online indexing strategies try to avoid these costly preparation cost by first monitoring the query workload and its performance when processing the queries. New indexes will be built/updated (or old indexes will be dropped) once certain thresholds are reached. The downside is that the index updates may severely affect the query processing performance and existing indexes may be outdated or become ineffective as soon as the query workload changes and thus queries may need to be answered without index support until one of the next thresholds is reached.

In dealing with large scientific data in dynamic workload, efficient computation becomes an important factor in reducing the processing costs as well as the preparation costs. One may want to process only the necessary things for the query at hand, that is, to do *just enough*. That is the philosophy of the *Database Cracking*, a recent indexing strategy [53]. Cracking is designed to work under the assumption that no idle time and no prior workload knowledge required.

Cracking uses the user queries as *advice* to refine the physical datastore and its indexes. The cracking philosophy has the goal of lightweight processing and quick adaptation to user queries. That is, the response time rapidly improves as soon as the next query arrives. However, under a dynamic environment, this can backfire. Blindly following the user queries may create (cracker) indexes that are detrimental to the overall query performance. This robustness problem causes cracking fails to adapt and consumes significantly far more resources than needed, and turn it into a big data problem.

We propose *stochastic cracking* to relax the philosophy by investing some resources to ensure that future queries continue to improve on its response time and thus able to maintain an overall efficient, effective, and robust cracking under dynamic and unpredictable query workloads. To achieve this robustness property, stochastic cracking looks at the property of the underlying data as well instead of blindly following the user query entirely. Stochastic cracking maintains the sub-linear complexity in query processing and conforms to the original cracking interface, thus, can be used as a drop in replacement for the original cracking.

In this thesis, we propose several cracking algorithms and present extensive comparisons among them. Our stochastic cracking algorithm variants manage to outperform the original cracking by two orders of magnitude faster on a real dataset and real dynamic query workload while the offline indexing is still halfway through preparing the indexes.

1.1.3 Large Graph Processing

Graphs from the Internet such as the World Wide Web and the online social networks are extremely large. Analyzing such graphs is a big data problem. Typically, such large graphs are stored and processed in a distributed manner as it is more economical to do so rather than in a centralized manner (e.g., using a super computer with terabytes of memory and thousands of cores). However, running graphs algorithms that have quadratic runtime complexity or more will quickly become impractical on such large graphs as the available resources (i.e., the number of machines) only scales linearly as the graph size. To solve this big data problem in practice, one must invent more effective new solutions without compromising the result quality.

Fortunately, many large real-world graphs have been shown to exhibit small-world network (SWN) properties (in particular, they have been shown to have small diameter) and robust. As we shall see in this thesis, we can exploit the inherent properties of the SWN, in particular, the small diameter property and robustness against edge removal, to redesign a quadratic graph algorithm such as the *Maximum-Flow* (max-flow) algorithm into new parallel and distributed algorithms. We show empirically that it has a linear runtime complexity in terms of the graph size. The max-flow problem is a classical graph problem that has many useful applications in the World Wide Web as well as in the online social networks such as finding spam sites, building content voting system, discovering communities, etc.

The performance and scalability of the new algorithms depend on the processing framework. As of this writing, the existing specialized distributed graph processing frameworks based on Google Pregel are still under development[1]. Therefore, most of current researches on large graph processing are built on top of the *MapReduce* framework which has become de facto standard for processing large-scale data over thousands of commodity machines.

In this thesis, we redesigned, implemented, and evaluated the existing max-flow algorithms (namely the Push-Relabel algorithm and the Ford-Fulkerson method) on the MapReduce framework. Implementing these *non trivial* graph algorithms on the MapReduce framework has its own challenges. The algorithms must be represented in the form of *stateless* MAP and REDUCE functions and the data must be represented in *records* of $\langle key, value \rangle$ pair. The algorithm must work in a local (or distributed) manner (i.e., only use the information in a local *record*). Moreover, since the cost of fetching the data (from disks and/or network) far outweigh the costs of computing the data (applying the MAP or

[1]Pregel is proprietary to Google while Apache Giraph and Hama are still in incubator phase.

REDUCE functions), the algorithms must be tailored to a new cost model. We describe the design, parallelization and optimizations needed to effectively compute max-flow for the MapReduce framework. We believe that these optimizations are useful as design patterns for MapReduce based graph algorithms as well as specialized graph processing frameworks such as Pregel. Our new highly parallel MapReduce-based algorithms that exploit the small diameter of the graph are able to compute max-flow on a subset of the Facebook social network graph with 411 million vertices and 31 billion edges using a cluster of 21 machines in reasonable time.

1.2 The Structure of this Thesis

The chapters are organized as follows:

- Chapter 1, we discuss Big Data problems and introduce the three problems and their common solutions in terms of (sub)linear complexity, stochastic/non-deterministic algorithm, robustness, and exploitation of the inherent properties of the data.
- Chapter 2, we discuss how to utilize stochastic local-search together with the optimal algorithm into an effective and efficient segmentation algorithm.
- Chapter 3, we discuss how stochasticity helps to make database cracking robust under dynamic and unpredictable environment.
- Chapter 4, we discuss strategies to exploit the small diameter property of the graph and transform a classic maximum flow algorithm into a highly parallel and distributed algorithm by leveraging the MapReduce framework.
- Chapter 5, we conclude our thesis and summarize the important lessons learned.

1.3 List of Publications

During the PhD candidature at School of Computing, National University of Singapore, the author has published the following works which are related to the thesis (in chronological order):

1. Felix Halim, Yongzheng Wu and Roland H.C. Yap. Security Issues in Small World Network Routing. In the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2008). IEEE Computer Society, 2008.

2. Felix Halim, Yongzheng Wu, and Roland H.C. Yap. Small world networks as (semi)-structured overlay networks. In Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, 2008.

3. Felix Halim, Yongzheng Wu, and Roland H.C. Yap. Wiki credibility enhancement. In the 5th International Symposium on Wikis and Open Collaboration, 2009.

4. Felix Halim, Panagiotis Karras, and Roland H.C. Yap. Fast and effective histogram construction. In 18th ACM Conference on Information and Knowledge Management (CIKM), 2009. (**best student paper runner-up**)

5. Felix Halim, Panagiotis Karras, and Roland H.C. Yap. Local search in histogram construction. In 24th AAAI Conference on Artificial Intelligence, July 2010.

6. Felix Halim, Yongzheng Wu and Roland H.C. Yap. Routing in the Watts and Strogatz Small World Networks Revisited. In Workshops of the 4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO Workshops 2010), 2010.

7. Felix Halim, Roland H.C. Yap and Yongzheng Wu. A MapReduce-Based Maximum-Flow Algorithm for Large Small-World Network Graphs. In the 2011 IEEE 31th International Conference on Distributed Computing Systems (ICDCS'11), IEEE Computer Society, 2011.

8. Felix Halim, Stratos Idreos, Panagiotis Karras and Roland H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. In the 38th Very Large Databases Conference (VLDB), Istanbul, 2012.

9. Goetz Graefe, Felix Halim, Stratos Idreos, Harumi Kuno and Stefan Manegold. Concurrency Control for Adaptive Indexing. In the 38th Very Large Databases Conference (VLDB), Istanbul, 2012.

The following are the other publications the author has been involved in during his doctoral candidature (in chronological order):

1. Steven Halim, Roland H. C. Yap, Felix Halim. Engineering Stochastic Local Search for the Low Autocorrelation Binary Sequence Problem. International Conference on Principles and Practice of Constraint Programming, 2008.

2. Felix Halim, Rajiv Ramnath, YongzhengWu, and Roland H.C. Yap. A lightweight binary authentication system for windows. International Federation for Information Processing Digital Library, Trust Management II, 2008.

3. Yongzheng Wu, Sufatrio, Roland H.C. Yap, Rajiv Ramnath, and Felix Halim. Establishing software integrity trust: A survey and lightweight authentication system for windows. In Zheng Yan, editor, Trust Modeling and Management in Digital Environments: from Social Concept to System Development, chapter 3. IGI Global, 2009.

4. Yongzheng Wu, Roland H.C. Yap, and Felix Halim. Visualizing Windows system traces. In Proceedings of the 5th International Symposium on Software visualization (SOFTVIS'10), ACM, 2010.

5. Suhendry Effendy, Felix Halim and Roland Yap. Partial Social Network Disclosure and Crawlers. In Proceedings of the International Conference on Social Computing and its Applications (SCA 2011), IEEE, 2011. (**best student paper**)

6. Suhendry Effendy, Felix Halim and Roland Yap. Revisiting Link Privacy in Social Networks. In Proceeding of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY12), ACM, 2012.

Chapter 2

Sequence Segmentation

A *segmentation* aims to approximate a data sequence of values by piecewise-constant line segments, creating a small synopsis of the sequence that effectively capture the basic features of the underlying data. Sequence segmentation has wide area of applications. In time series databases, it has been used for context recognition [52], indexing and similarity search [21]; in bio-informatics for DNA [79] or genome segmentation [95]; in database systems for data distribution approximation [61], intermediate join results approximation by a query optimizer [59, 84], query processing approximation [90, 20], and *point* and *range queries* approximation [45]; the same form of approximation is used in knowledge management applications as in decision-support systems [11, 63, 106]. An overview of the area from a database perspective is provided in [60, 61].

In all cases, a segmentation algorithm is employed in order to divide a given data sequence into a given budget of consecutive *buckets* or *segments* [52]. All values in a segment are approximated by a single representative. Both these representative values, and the bucket *boundaries* themselves, are chosen so as to achieve a low value for an error metric in the overall approximation. Depending on the application domain, the same approximate representation of a data sequence is called a *histogram* [61], a *segmentation* [104], a *partitioning*, or a *piecewise-constant approximation* [21].

The importance of sequence segmentation becomes more apparent in the context of mobile devices [52]. Recent advances in micro-sensor technology raises interesting challenges in how to effectively analyze large data sequence in such devices where computation and communication bandwidth are scarce resources [10]. In such limited resources environment, sequence segmentation becomes a big data problem. An optimal segmentation derived by a quadratic dynamic-programming (DP) algorithm that recursively examines all possible solutions [15, 65] is impractical. Thus, heuristic approaches [92, 65] are employed in practice.

Recent research has revisited the problem from the point of view of approxi-

mation algorithms. Guha et al. proposed a suite of approximation and streaming algorithms for histogram construction problems [44]. Of the algorithms proposed in [44], the AHistL-Δ proves to be the best for offline approximate histogram construction. In a nutshell, AHistL-Δ builds on the idea of approximating the error function itself, while pruning the computations of the DP algorithm. Likewise, Terzi and Tsaparas recently proposed DnS, an offline approximation scheme for sequence segmentation [104]. DnS divides the problem into subproblems, solves each of them optimally, and then utilizes DP to construct a global solution by merging the segments created in the partial solutions.

In solving big data problems, one must be wise in spending resources. The results should be commensurate with the resources spent. Despite their theoretical elegance, the approximation algorithms proposed in previous research do not always resolve the tradeoffs between time complexity and histogram quality in a satisfactory manner. The running time of these algorithms can approach that of the quadratic-time DP solution. Still, the quality of segmentation they achieve can substantially deviate from the optimal. Previous research has not examined how the different approximation algorithms of [44] and [104] compare to each other in terms of efficiency and effectiveness.

In this chapter, we propose a middle ground between the theoretical elegance of approximation algorithms on the one hand, and the simplicity, efficiency, and practicality of heuristics on the other. We develop segmentation algorithms that run in linear complexity in order to scale to large data sequence. While these algorithms do not provide approximation guarantees with respect to the optimal solution, they produce better segmentation quality than the existing algorithms. We employ stochastic features by way of a local search algorithm. It results in a segmentation which is very effective in extracting the characteristics of the underlying data sequence. Our stochastic local search consistently produces solutions where its segmentation positions are near if not the same to optimal segmentation positions and these solutions can be recombined into a significantly better solution without sacrificing the linear runtime complexity. We demonstrate that our solution is scalable and provides the best tradeoff between runtime versus quality that allows them to be employed in practice, instead of the currently used heuristics, when dealing with the segmentation of very large data sets under limited resources. We conduct the first, to our knowledge, experimental study of state-of-the-art optimal, approximation, and heuristic algorithms for sequence segmentation (or histogram construction). This study demonstrates that our algorithms vastly outperform the guarantee-providing approximation schemes in terms of running time, while achieving comparable or superior approximation accuracy.

Our work local search algorithm and the hybrid algorithms that use it as sampling is published in [47] while our analysis on local search for histogram construction is published in [48].

2.1 Problem Definition

Given a data sequence $\mathbf{D} = \langle d_0, d_1, \ldots, d_{n-1} \rangle$ of length n. We define a segmentation of $\mathbf{D}$ as $\mathbf{S} = \langle b_0, b_1, \ldots, b_B \rangle$ where $b_i \in [0, n-1]$. b_i denote the *boundary* positions of the data sequence. The first boundary b_0 and the last boundary b_B are fixed at position $b_0 = 0$ and $b_B = n$. The intervals $[b_{i-1}, b_i - 1] \mid i \in [1, B]$ are called *buckets* or *segments*. Each segment is attributed a representative value v_i, which approximate all values d_j where $j \in [b_{i-1}, b_i - 1]$. Figure 2.1 gives the illustration.

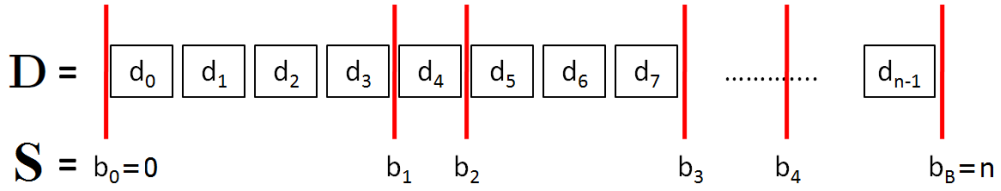


Figure 2.1: A segmentation $\mathbf{S}$ of a data sequence $\mathbf{D}$

The goal of a *segmentation algorithm* is to find boundary positions that achieve a low approximation error for the error metric at hand. A useful metric is the *Euclidean* error which in practice works on the sum-of-squared-errors (SSE). Previous studies [65, 31, 71, 43, 93, 72, 73, 74, 69, 70] have generalized their results into wider classes of *maximum*, *distributive*, *Minkowski-distance*, and *relative-error* metrics. Still, the Euclidean error remains an important error metric (and the most well known) for several applications, such as database query optimization [62], context recognition [52], and time series mining [21].

For a given target error metric, the representative value v_i of a bucket that minimizes the resulting approximation error is straightforwardly defined as a function of the data values in the bucket. For the *average absolute error* the best v_i is the *median* of the values in the interval [104]; for the *maximum absolute error* it is the mean of the maximum and minimum value in the interval [75]; an analysis of respective relative-error cases is offered in [45]. For the Euclidean error that concerns us, the optimal value of v_i is the *mean* of values in the interval [65].

2.2 The Optimal Segmentation Algorithm

The $O(n^2B)$ dynamic-programming (DP) algorithm that constructs an optimal segmentation, called V-Optimal, under the Euclidean error metric is a special case of Bellman's general line segmentation algorithm [15]. This was first presented by Jagadish et al. [65] and optimized in terms of space-efficiency by Guha [42]. Its basic underlying observation is that the optimal b-segmentation of a data sequence $\mathbf{D}$ can be recursively derived given the optimal $(b-1)$-segmentations of all prefix sequences of $\mathbf{D}$. Thus, the minimal sum-of-squared-errors (SSE) $E(i,b)$ of a b-bucket segmentation of the prefix sequence $\langle d_0, d_1, \ldots, d_i \rangle$ is recursively expressed as:

$$E(i,b) = \min_{b \leq j < i} \{E(j, b-1) + \mathcal{E}(j+1, i)\} \tag{2.1}$$

where $\mathcal{E}(j+1,i)$ is the minimal SSE for the segment $\langle d_{j+1}, \ldots, d_i \rangle$. This error is easily computed in $O(1)$ based on a few pre-computed quantities (sums of squares and squares of sums) for each prefix [65]. Thus, this algorithm requires a $O(nB)$ tabulation of minimized error values $E(i,b)$ along with the selected optimal last-bucket boundary positions j that correspond to those optimal error values. As noted by Guha, the space complexity is reducible to $O(n)$ by discarding the full $O(nB)$ table; instead, only the two *running* columns of this table are stored. The *middle bucket* of each solution is kept track of; after the optimal error is established, the problem is divided in two half subproblems and the same algorithm is recursively re-run on them, until all boundary positions are set [42]. The runtime is significantly improved by a simple pruning step [65]; for given i and b, the loop over (decreasing) j that searches for the min value in Equation 2.1 is broken when $\mathcal{E}(j+1,i)$ (non-decreasing as j decreases) exceeds the running minimum value of $E(i,b)$.

Unfortunately, the quadratic time complexity of V-Optimal renders it inapplicable in most real-world applications. Thus, several works have proposed approximation schemes [33, 34, 44, 104].

2.3 Approximations Algorithms

Recent research has revisited the segmentation problem [104] (or histogram construction problem in database context [44]) from the point of view of approximation algorithms. This section details these approximation approaches.

2.3.1 AHistL − Δ

Guha et al. have provided a collection of approximation algorithms for the histogram construction problem. Out of them, AHistL-Δ is their algorithm of choice for offline histogram construction [44].

The basic observation underlying the AHistL-Δ algorithm is that the $E(j, b-1)$ function in Equation 2.1 is a non-decreasing function of j, while its counterpart function $\mathcal{E}(j+1, i)$ is a non-increasing function of j. Thus, instead of computing the entire tables of $E(j, b)$ over all values of j, this non-decreasing function is approximated by a staircase histogram representation - that is, a histogram in which the representative value for a segment is the highest (i.e., the rightmost) value in it. In effect, only a few representative values of $E(j, b-1)$, i.e., the end values of the staircase intervals, are used. Moreover, the segments of this staircase histogram themselves are selected so that the value of the $E(j, b)$ function at the right-hand end of a segment is at most $(1+\delta)$ times the value at the left-hand end, where $\delta = \frac{\epsilon}{2B}$. The recursive formulation of Equation 2.1 remains unchanged, with the difference that the variable j now only ranges over the endpoints of intervals in this staircase histogram representation of $E(j, b)$. Figure 2.2 illustrates how space can be saved by approximating $E(i, b)$.

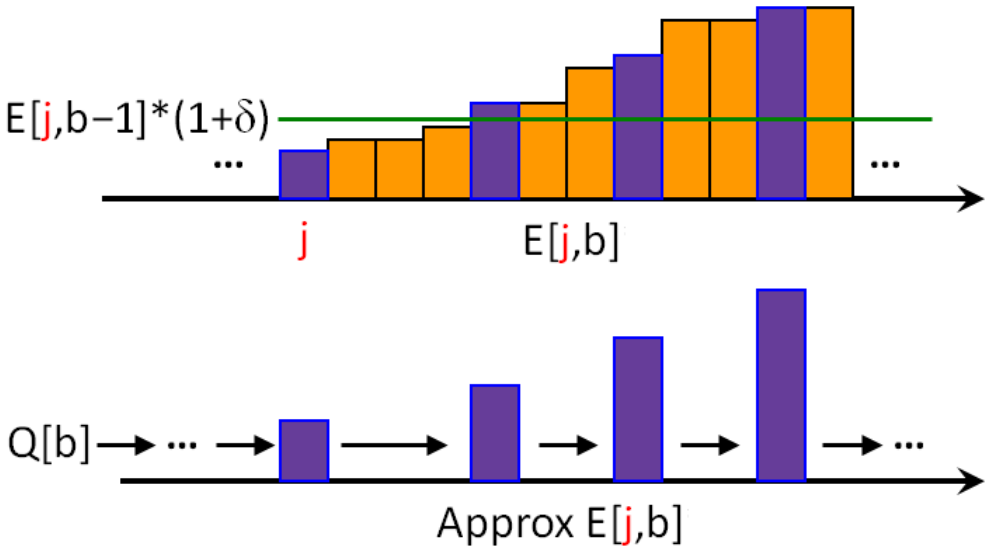


Figure 2.2: AHistL − Δ - Approximating the $E(j, b)$ table

On top of this observation, the AHistL-Δ algorithm adds the further insight that any $E(j, b)$ value that exceeds the final SSE of the histogram under construction (or even an approximate version of that final error) *cannot* play a role in the eventual solution. Since $E(j, b)$ form partial contributions to the *aggregate* SSE, only values smaller than the final error Δ contribute to the solution. Thus, if we knew the error Δ of the V-Optimal histogram in advance, then we could eschew

the computation of any $E(j, b)$ value that exceeds it. Since Δ in not known in advance, we can still work with estimates of Δ in a binary-search fashion.

The AHistL-Δ algorithm combines the above two insights. In effect, the problem is decomposed in two parts. The *inner* part returns a histogram of error less than $(1+\epsilon)\Delta$, under the assumption that the given estimate Δ is correct, i.e., there exists a histogram of error Δ; the *outer* part searches for such a well-chosen value of Δ. The result is an $O(n + B^3(\log n + \epsilon^{-2}) \log n)$-time algorithm that computes an $(1+\epsilon)$-approximate B-bucket histogram.

2.3.2 DnS

Terzi and Tsaparas correctly observed that a quadratic algorithm is not an adequately fast solution for practical sequence segmentation problems [104]. As an alternative to the V-Optimal algorithm, they suggested a sub-quadratic constant-factor approximation algorithm.

The basic idea behind this *divide and segment* (DnS) algorithm is to divide the overall segmentation problem into smaller subproblems, solve those subproblems optimally, and then combine their solutions. The V-Optimal algorithm serves as a building block of DnS. The problem sequence is arbitrarily partitioned into smaller subsequences. Each of those is optimally segmented using V-Optimal. Then, the derived segments are treated as the input elements themselves, and a segmentation (i.e., local merging) of them into B larger buckets is performed using the V-Optimal algorithm again. A thorough analysis of this algorithm demonstrates an approximation factor 3 in relation to the optimal Euclidean error error, and a worst-case complexity of $O(n^{4/3}B^{5/3})$, assuming that the original sequence is partitioned into $\chi = \left(\frac{n}{B}\right)^{2/3}$ equal-length segments in the first step. The recursive application of the DnS results in $O(nB^3 loglogn)$ for $\chi = \sqrt{n}$ which is slower than the AHistL-Δ.

2.4 Heuristic Approaches

Past research has also proposed several heuristics for histogram construction. Some of these heuristics are relatively brute-force segmentations; this category includes methods such as the end-biased [62], equi-width [76], and equi-depth [89, 87] heuristics.

A more elaborate heuristic is the MaxDiff [92] method. According to this method, the $B-1$ points of highest difference between two consecutive values in the original data set are selected as the boundary points of a B-bucket histogram. Its time complexity is $O(n \log B)$, i.e., the cost of inserting n items into a priority

queue of B elements. Poosala and Ioannidis conducted an experimental study of several heuristics employed in database applications and concluded that the MaxDiff-histogram was "probably the histogram of choice". Matias et al. used this method as the conventional approach to histogram construction for selectivity estimation in database systems, in comparison to their alternative proposal of wavelet-based histograms [84].

Jagadish et al. [65] have suggested an one-dimensional variant of the multidimensional MHIST heuristic proposed by Poosala and Ioannidis [91]. This is a greedy heuristic that repeatedly selects and splits the bucket with the highest SSE, making B splits. The same algorithm is mentioned by Terzi and Tsaparas by the name Top-Down [104]; a similar algorithm has been suggested in the context of multidimensional anonymization by LeFevre et al. [78]. Its worst-case time complexity is $O(B(n + \log B)$, i.e., the cost to create a heap of B items while updating affected splitting costs at each step. In the pilot experimental study of [65], MaxDiff and MHIST turn out to be the heuristics of choice; it is observed that the former performs better on more spiked data, while the latter is more competitive on smoother data.

2.5 Our Hybrid Approach

Under big data context where the amount of available resources is limited, a segmentation algorithm must give a good justification on the resources spent. It should provide a satisfactory tradeoff between efficiency and accuracy, thus providing a significant advantage with respect to both the optimal but not scalable V-Optimal and to fast but inaccurate heuristics. Approximation schemes with time super-linear in n and/or B may not achieve this goal. There is a need for new approaches that can provide the best of the both world by having near-optimal segmentation quality and near-linear runtime complexity. In this section, we proposed our algorithms based on local search combined by the existing DP to produce near ideal segmentation algorithm. Although our local search and existing heuristics are both greedy-based algorithms, there are important differences. Our local search algorithms employ iterative improvement and stochasticity. In contrast, both MaxDiff and MHIST derive a solution in one shot and never modify a complete B-segmentation they have arrived at.

2.5.1 Fast and Effective Local Search

We first describe a basic local search algorithm called GDY. It starts with an *ad hoc* segmentation $\mathcal{S}_0$ and makes local moves which greedily modify boundary

positions so as to reduce the total $\mathcal{L}_2$ error. $\mathcal{S}_0$ can be randomly created, or it can be that of a simple heuristic, for example an equi-width histogram [76]. Each local move has two components. First a segment boundary whose removal incurs the *minimum* error increase is chosen. As this decreases the number of segments by one, a new segment boundary is added back by splitting the segment which gives the *maximum* error decrease. Note that expanding or shrinking a segment by moving its boundary is a special case of this local move when the same segment is chosen for removal and splitting. A local minimum on the total error is reached when no further local moves are possible. Figure 2.3 gives an illustration of a local move.

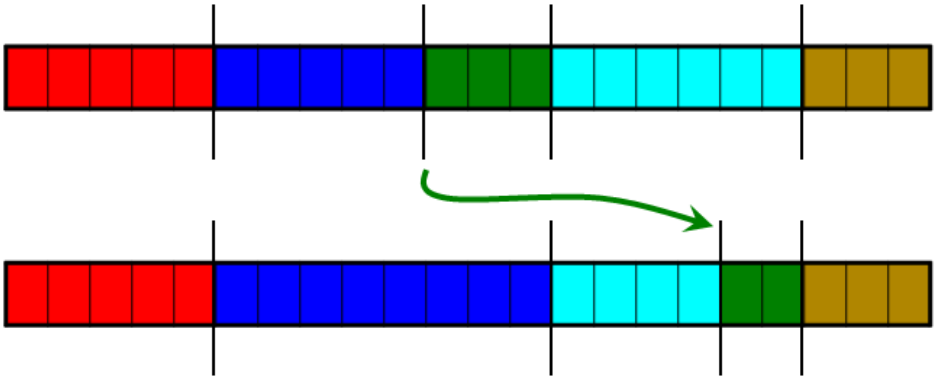


Figure 2.3: Local Search Move

We define boundary positions b_i where $1 \leq i \leq B-1$ as *movable* boundary positions. As described in Section 2.1, the first and last boundary positions (b_0 and b_B) are fixed. To ensure efficient local moves, we keep a min-heap H^+ of *movable* boundary positions with their associated potential *error increase*, and a max-heap H^- of all *movable* segments with the potential *error decrease*. The time complexity of GDY is $O\left(M\left(\frac{n}{B}+\log B\right)\right)$, where M is the number of local moves, $\frac{n}{B}$ for the (average) cost of calculating the optimal split position for a newly created segment after each move, and $\log B$ for the overhead of selecting the best-choice boundary to remove and segment to split using the heaps.

Figure 2.4 gives the GDY algorithm. The GDY algorithm first takes an input of a data sequence $\mathbf{D}$ and output a segmentation $\mathbf{S}$ of B segments. An initial segmentation S_0 is created with randomized $B-1$ movable segment boundaries (Line 1). Populate all $B-1$ movable boundaries into a min-heap H^+ and a max-heap H^-. Do improving local search move until stuck (Line 3-12). We want to move from solution S_{i-1} to a new solution S_i (Line 4). We take the boundary G that results in the minimum error increase Δ^+E_i from H^+ (Line 5). We remove G from S_i and update the error increase and error decrease in the heaps. Note that the heaps do not support delete operations, however, to simulate the "update"

Algorithm GDY(B)
Input: space bound B, n-data sequence $\mathbf{D} = [d_0, \ldots, d_{n-1}]$
Output: a segmentation $\mathbf{S}$ of B segments
1. S_0 = randomized initial segmentation
2. $i = 0$; Populate H^+ and H^- with S_i;
3. **while** (H^+ is not empty)
4. $i = i + 1$; $S_i = S_{i-1}$;
5. G = take boundary with minimum $\Delta^+ E_i$ from H^+;
6. Remove G from S_i and update H^+ and H^-;
7. P = take segment with maximum $\Delta^- E_j$ from H^-;
8. **if** ($\Delta^+ E_i - \Delta^- E_j >= 0$)
9. Undo steps 6 and 7; // G is discarded
10. **else**
11. Split segment P, add new boundary to S_i;
12. Update partitions' costs in H^+ and H^-;
13. **return** S_i;

Figure 2.4: GDY algorithm

we can just insert the new boundary and its cost to the heaps and discard invalid boundary positions upon extracting from the heaps. We can do this since we know the exact boundary positions of the current segmentation S_i. Thus, the update is still of order $O(logB)$ (Line 6). We then take the segment P with maximum error decrease $\Delta^- E_j$ from H^- (line 7). If the error increase is bigger than the error decrease (Line 8), then we undo the steps we did on line 6 and 7. This will result in one less element for H^+ but the size of H^- stay constant. If this keep happening, then after a number of local move, H^+ will be empty and the GDY algorithm stuck in a local optima (Line 3) and the current segmentation S_i is returned (Line 13). Otherwise, if the error increase is less than the error decrease (Line 10), then we split the segment P in S_i adding a new boundary inside P (Line 11) and the heaps are updated accordingly (Line 12).

According to our experimental analysis, GDY produces segmentation with better $\mathcal{L}_2$ error with on par performance to existing heuristic approaches under variety of datasets. In the next section, we present a way to significantly boost the quality (lower the $\mathcal{L}_2$ error) without sacrificing much the performance advantages.

2.5.2 Optimal Algorithm as the Catalyst for Local Search

The proposed approximation algorithms AHistL $-\ \Delta$ [44] and DnS [104] aim to eschew part of the DP computation without altogether discarding the dynamic-programming (DP) itself. AHistL-Δ tries to carefully discard from the DP recursion those candidate boundary positions whose error contribution can be *ap-*

proximated by that of their peers [44]. Likewise, DnS attempts to acquire a set of *samples* (i.e., candidate boundary positions) by running the DP recursion itself in a small scale, within each of the χ subsequences it creates [104], ending up with χB samples. Thus, DP is applied in a two-level fashion, both at a micro-scale, within each of the χ subsequences, and at a macro-scale, among the derived boundaries themselves. While this approach allows for an error guarantee, it unnecessarily burdens what is anyway a suboptimal algorithm with super-linear time complexity. Likewise, in its effort to provide a *bounded approximation* of every error quantity that enters the problem, AHistL-Δ ends up with an $O(B^3(\log n + \epsilon^{-2}) \log n)$ time complexity factor that risks exceeding even the runtime of V-Optimal itself in practice.

The DP algorithm that aims to discover a good segmentation does not need to examine every possible boundary position per se; it can constrain itself to a limited set of candidate boundary positions that are deemed to be likely to participate in an optimal solution. Thus, the question is how to effectively and efficiently identify a set of such candidate boundary positions, which we call *samples*. We define a sample of candidate boundary position *good* if it belongs to at least one of the partition sets of an optimal segmentation and *bad* otherwise.

We observed that one run of GDY often finds at least 50% of good samples. Because of the stochasticity of the GDY, if we perform another run of GDY, it will produce a slightly different set of partitions and find a slightly different set of 50% of good samples. We also observed that running a small number iteration of GDY produce enough good samples that cover most if not all candidate boundary positions that participate in a partitions set of an optimal segmentation. This turns out to be a very efficient and effective way of generating good samples. *All* boundary positions collected in this fashion are themselves participants in a B-segmentation of the input sequence that GDY could not improve further; thus, they are reasonable candidates for an *optimal* B-segmentation. We emphasize that this approach to sample collection contrasts to the one followed in DnS [104]; the DnS samples do not *themselves* participate in a global B-segmentation, but only in local B-segmentations of subsequences. Thus, even though that methodology allows for the computation of an elegant approximation guarantee, most of the samples it works with are unlikely to participate in an optimal B-segmentation. An important point is that unlike DnS, our sampling process is non-uniform and results in more samples at subspaces where more segments may be needed, thus reducing error. A similar observation holds for AHistL-Δ. This algorithm endeavors to discard computations related to those boundary positions that have produced error upper-bounded by that of one of their neighbors in an examined subproblem; however, it does not aim to work on boundary positions

Algorithm GDY_DP(B, I)
Input: bound B, number of runs I, n-data sequence $[d_0, \ldots, d_{n-1}]$
Output: a segmentation **S** of B segments
1. $\mathcal{S}$ = empty set of sample boundaries
2. **loop** (I times)
3. $\mathcal{P}$ = GDY(B); // run GDY with random initialization
4. $\mathcal{S} = \mathcal{S} \cup \mathcal{P}$;
5. **return** DP($\mathcal{S}, B$);

Figure 2.5: GDY_DP algorithm

that are more likely to *participate in the optimal solution* per se.

Utilizing GDY as sample generator, we introduce a new algorithm, GDY_DP. In case B is less than $\sqrt{n}$, we run I iterations of GDY until we collect up to $O(\sqrt{n})$ samples. We expect that a large percentage of the partitions in the candidate sample set are also in an optimal solution. Then, we run the V-Optimal DP segmentation algorithm on this set of samples, in order to select a subset of B out of them that define a minimum-error segmentation. Since the sample set has at most IB partitions, it is still $O(B)$, as such selecting the best B out of the set using dynamic programming costs $O(B^3)$ which gives a total runtime of $O(nB)$ as $B \leq \sqrt{n}$. Thus, dynamic programming is used to provide a high-quality segmentation without sacrificing the linear time complexity of GDY. Figure 2.5 presents a pseudo-code for this GDY_DP algorithm.

2.5.3 Scaling to Very Large n and B

When B is larger than $\sqrt{n}$, GDY_DP loses its linear-in-n character, but still runs faster than both DnS and AHistL-Δ. Now, we can no longer use V-Optimal to recombine the samples to get a guaranteed improvement on the local search segmentation, while also maintaining a linear time complexity in n because the time complexity of the DP step in GDY_DP becomes too large. We introduce the GDY_BDP algorithm, a batch-processing version of GDY_DP to handle this case.

The idea is to process only $\sqrt{n}$ samples at a time. That is, we divide the sorted sequence of collected samples into subsequences of at most $\sqrt{n}$ consecutive samples; we run separate DP segmentations on each those; and we augment the results into a global B-segmentation. Our approach is reminiscent of the suggested piecewise application of a summarization scheme in [74]. However, it combines the batched or piecewise processing approach with a sample selection mechanism, as in [104]. Thus, it contains the size of the problem in two ways: (i) it examines only selected samples instead of the whole data; and (ii) it processes the data in batches.

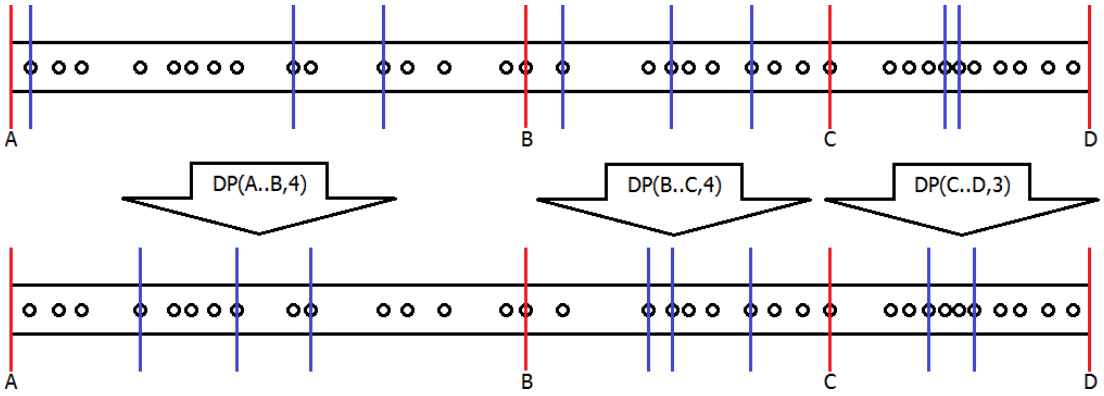


Figure 2.6: GDY_BDP Illustration

In more detail, we use an *auxiliary* segmentation $\mathcal{A}$, which is provided by a single run of GDY. For each group $\mathcal{G}$ of roughly $\sqrt{n}$ consecutive samples, we find two *dividers* l_a and r_a, among the boundaries in $\mathcal{A}$, that enclose $\mathcal{G}$. We then run V-Optimal on the set $\mathcal{G}$, so as to produce as many boundaries between l_a and r_a as the segmentation $\mathcal{A}$ has allocated in this interval. Thus, the number of boundaries in the derived segmentation is kept appropriate, but their positions are bettered within each $\sqrt{n}$-boundary subinterval. The intuition is that we improve on the boundary positions derived by GDY in $\mathcal{A}$, but we do so in a more sophisticated manner than just greedily moving one boundary at a time. Thus, the quality of the segmentation is improved.

In each group $\mathcal{G}$ we are dealing with $\sqrt{n}$ samples, while there are $O(I \times B)$ samples in total to deal with, where I the constant pre-selected number of GDY runs that produces them. Thus, the are $O(\frac{IB}{\sqrt{n}})$ groups of $\sqrt{n}$ samples each, hence the algorithm performs as many separate DP runs. Each run chooses a fraction of the $\sqrt{n}$ samples in group $\mathcal{G}$, performing an $O(\sqrt{n}^3) = O(n\sqrt{n})$ DP segmentation over them. Thus, the overall worst-case time complexity of the GDY_DP algorithm is $O(\frac{IB}{\sqrt{n}}\sqrt{n}^3) = O(InB) = O(nB)$.

Figure 2.6 illustrates the GDY_BDP algorithm. The upper figure is a segmentation $\mathcal{A}$ produced by one GDY. The small circles are the samples produced by some I GDY iterations. The samples are divided into 3 groups, each with roughly of $\sqrt{n}$ size. The DP are performed to each group producing the same number of partitions with the number of partitions from $\mathcal{A}$ that resides in that group. For example, the first group contains samples from A to B and the DP is performed to produce 4 segments since there are 4 segments in $\mathcal{A}$ that resides in $[A..B]$. Similarly for the second group that contains samples from $[B..C]$. The last group only produces 3 segments. Thus, the DP is used to improve the segmentations inside each group (or batch). Note that the partition at position A, B, C, D are not optimized by the DP. They are *sacrificed* as *breakpoints*.

Algorithm GD_Batched_DP(B, I)
Input: space bound B, number of GDY runs I, n-data sequence $[d_0, \ldots, d_{n-1}]$
Output: a segmentation **S** of B segments
1. $\mathcal{S}$ = collect *samples* by running randomized GDY(B) I times;
2. $\mathcal{A}$ = GDY(B); // an *auxiliary* solution to be improved
3. $\mathcal{S} = \mathcal{S} \cup \mathcal{A}$;
4. $l_s = 0$; // left group divider in $\mathcal{S}$
5. $l_a = 0$; // left group divider in $\mathcal{A}$
6. **while** $(l_s + 1 < |\mathcal{S}|)$
7. $r_s = \min\{|\mathcal{S}| - 1, l_s + \sqrt{n}\}$; // right divider in $\mathcal{A}$
8. r_a = first boundary in $\mathcal{A}$ after r_s; // right divider in $\mathcal{S}$
9. r_s = matching boundary of r_a in $\mathcal{S}$; // adjusted right divider in $\mathcal{A}$
10. $\bar{B} = r_a - l_a$; $\mathcal{G} = \mathcal{S}[l_s..r_s]$;
11. optimalSubPars $= DP(\mathcal{G}, \bar{B})$
12. Replace $\mathcal{A}[l_a..r_a]$ with optimalSubPars
13. $l_s = r_s$; $l_a = r_a$; // update left index for next batch
14. **return** $\mathcal{A}$; // the improved aux solution

Figure 2.7: GDY_BDP algorithm

The rationale behind GDY_BDP is that, when the number of boundaries is large, selecting some breakpoints for them based on a simple GDY solution and then performing GDY_DP within the $\sqrt{n}$-sample intervals defined by these breakpoints does not hamper the overall quality too much. On the contrary, it confers near-optimal quality and allows for near-linear time efficiency. As we shall see, the quality achieved with GDY_BDP is always very close to that of GDY_DP, while GDY_BDP is much faster on large B. GDY_BDP also avoids a major loophole in the methodology of DnS. Namely, DnS *forces* each of the *arbitrarily selected* subsequences it works on to produce B samples, and then chooses from the total pool of samples. Thus, it does not pay attention to the actual form of the data. Cases where some data regions require much denser segmentation than other regions are not satisfactorily covered by DnS, but they are covered by GDY_BDP. To our knowledge, no other segmentation algorithm scales well in both the input size n and the number of segments B, while producing, as we will see, near-optimal quality in terms of error.

Figure 2.7 depicts a pseudo-code for this batch-DP algorithm GDY_BDP. First, the samples are collected by running I iterations of GDY, each with random initial segmentation (Line 1). We perform another run of GDY as an auxiliary $\mathcal{A}$ which will be improved later by DP in batches (Line 2). The auxiliary is also part of the samples (Line 3). We initialize the group batch divider for the samples and the auxiliary (Line 4 and 5). We iterate each group consisting of roughly

$\sqrt{(n)}$ samples (Line 6 - 13). We locate the right breakpoint for this group and the samples in the group (Line 7-9). We compute the number of partitions $\bar{B}$ of $\mathcal{A}$ that fall in the group (Line 10). The DP is performed on the group to produce $\bar{B}$ partitions based on the samples (Line 11). We replace the auxiliary partition positions in the group with the optimal subpartitions returned by the DP from that group (Line 12). We proceed to the next group batch divider (Line 13).

2.6 Experimental Evaluation

This section presents our extensive experimental comparison of known approximation and heuristic algorithms and the solutions we have proposed. In particular, we have compared the following algorithms:

- V-Optimal The optimal Euclidean-error histogram construction algorithm of Jagadish et al. [65]. This algorithm is a specialization of the line-segmentation technique proposed by Bellman [15]. We add the denotation 2 in the algorithm's name to indicate the fact that we utilize the simple pruning suggested in [65].
- DnS The algorithm for sequence segmentation suggested by Terzi and Tsaparas [104]. This algorithm receives no parameters apart from the number of subsequences it uses, for which the authors of [104] suggest an optimal value, $\chi = \left(\frac{n}{B}\right)^{2/3}$, which we employ. We utilize the pruning technique with this algorithm too, hence the denotation 2 in its name.
- AHistL-Δ The algorithm suggested by Guha et al. as the best choice for fast offline approximate histogram construction [44]. We have used two versions of AHistL-Δ, one for $\epsilon = 0.01$, and one for $\epsilon = 10$, in order to witness how AHistL-Δ resolves the quality-efficiency tradeoff when putting a premium on quality (former case) or efficiency (latter case). In the figures, we indicate the employed value of ϵ as a *percentage* value next to the legend name. Thus, AHistL1 denotes the variant of AHistL-Δ for which $\epsilon = 0.01$.
- MaxDiff The relatively elaborate early histogram heuristic which was seen as "probably the histogram of choice" by Poosala et al. [92].
- MHIST The one-dimensional adaptation suggested by Jagadish et al. [65] for the multidimensional heuristic proposed by Poosala and Ioannidis [91], and also mentioned by Terzi and Tsaparas by the name Top-Down [104].
- GDY Our simple greedy algorithm that iteratively moves a boundary from one position to another until it reaches a local optimum.
- GDY_DP Our hybrid algorithm that combines GDY and the DP algorithm. In the figures, we indicate the number of iterations of GDY that generates

the samples for GDY_DP in its legend name; for example, GDY_10DP denotes a GDY_DP that operates on samples generated by $I = 10$ GDY runs.

- GDY_BDP The variant of our enhanced greedy algorithm that performs the DP-based selection of optimal boundary-subsets in a *batched* manner, in order to maintain the complexity of the DP operation linear in n. As for GDY_DP, the number of runs of GDY that generates the employed samples is indicated in the legend name.

All algorithms were implemented with gcc 4.3.0, and experiments were run on a 2 Quad CPU Intel Core 2.4GHz machine with 4GB of main memory running a 64Bit version of Fedora 9. Table 2.1 summarizes the complexity requirements of these algorithms.

Method	**Time**	**Ref**
V-Optimal	$O\left(n^2B\right)$	[65]
DnS	$O\left(n^{4/3}B^{5/3}\right)$	[104]
AHistL-Δ	$O(n + B^3(\log n + \epsilon^{-2}) \log n)$	[44]
MaxDiff	$O(n \log B)$	[92]
MHIST	$O(B(n + \log B))$	[65]
GDY	$O\left(M\left(\frac{n}{B} + \log B\right)\right)$	this
GDY_DP	$O\left(nB\right)$ $(B < \sqrt{n})$	this
GDY_BDP	$O\left(nB\right)$	this

Table 2.1: Complexity comparison

Our quality assessment uses several real-world time series data sets. These data sets provide a common ground for the comparisons as some of these data are also used in [104, 44]. Table 2.2 presents the original provenance[1] of the data. We have created *aggregated* versions of these data sets, i.e., concatenated the time series in them to create a united, longer sequence. In the following figures, we denote these aggregated versions by the appellation "-a" in the captioned data set names.

Name	**Size**	**Provenance**
Balloon	2001 x 2	http://lib.stat.cmu.edu/datasets/
Darwin	1400 x 1	http://www.stat.duke.edu/~mw/ts_data_sets.html
Exrates	2567 x 12	http://www.stat.duke.edu/data-sets/mw/ts_data/all_exrates.html
Phone	1708 x 8	http://www.teco.edu/tea/datasets/phone1.xls
Shuttle	1000 x 6	http://www-aig.jpl.nasa.gov/public/mls/time-series/
Winding	2500 x 7	http://www.esat.kuleuven.ac.be/~tokka/daisydata.html
DJIA16K	16384 x 1	http://lib.stat.cmu.edu/datasets/djdc0093 (filtered)
Synthetic	100001 x 10	http://kdd.ics.uci.edu/databases/synthetic/synthetic.html

Table 2.2: Used data sets

[1] The data are also available at http://felix-halim.net/histogram/.

2.6.1 Quality Comparisons

We first direct our attention to a comparison of quality, as measured by the Euclidean error achieved as a function of the available space budget B. In the following figures, the dotted line presents the position of 10% off the optimal error (always achieved by V-Optimal), while the dash-dotted line is the position of 20% off the optimal.

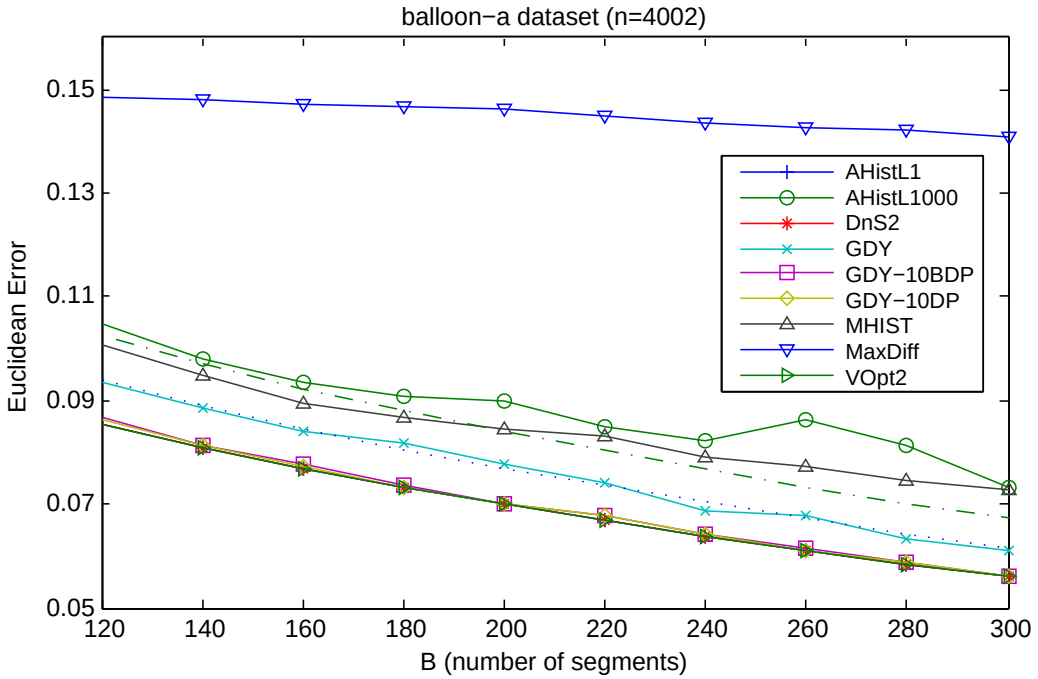


Figure 2.8: Quality comparison: Balloon

Figure 2.8 presents the results with the *aggregated* balloon data set for a selected range of $B = 120 \ldots 300$. We observe that all of DnS, AHistL-Δ for $\epsilon = 0.01$, GDY_DP, and GDY_BDP achieve practically indistinguishable near-optimal error. There are four outliers. The performance of GDY lies reliably along the 10%-off-optimal line; this is the best-performing outlier. The second outlier is MHIST; as expected, it does not produce histograms of near-optimal quality. Still, the variant of AHistL-Δ for $\epsilon = 10$ is even worse. This result is significant from the point of view of the tradeoff between quality and time-efficiency that AHistL-Δ achieves. We shall come back to it later. The MaxDiff heuristic had the worst performance.

Figure 2.9 shows the error results with the darwin data set for $B = 40 \ldots 200$. The picture exhibits a pattern similar to the previous one. Still, with this easier to approximate data set, GDY approaches the quality of the other near-optimal algorithms. Furthermore, the quality of MHIST now follows the 10%-off line. Now the MaxDiff heuristic achieves a smaller accuracy gap from the other contenders, but still has the worst quality. The low-quality version of AHistL-Δ is again an outlier with unreliable performance. The performance with other values of ϵ was

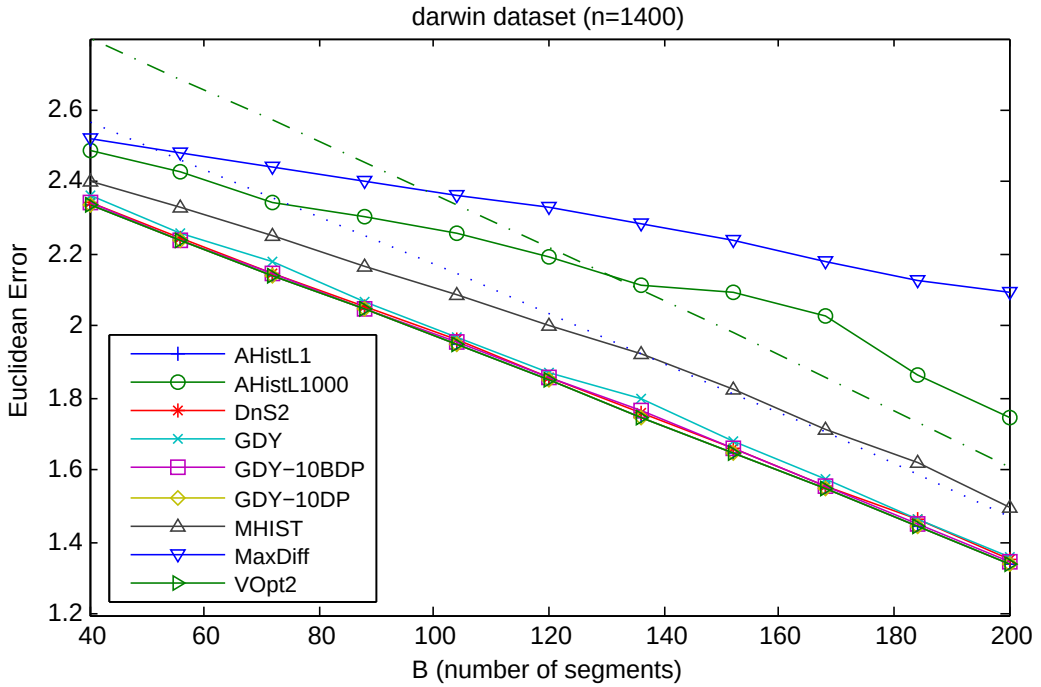


Figure 2.9: Quality comparison: Darwin

falling in between these two extremes. Naturally, the value of ϵ can be tuned so as to allow for quality that matches any of the other algorithms. We do not present these versions in order to preserve the readability of the graph.

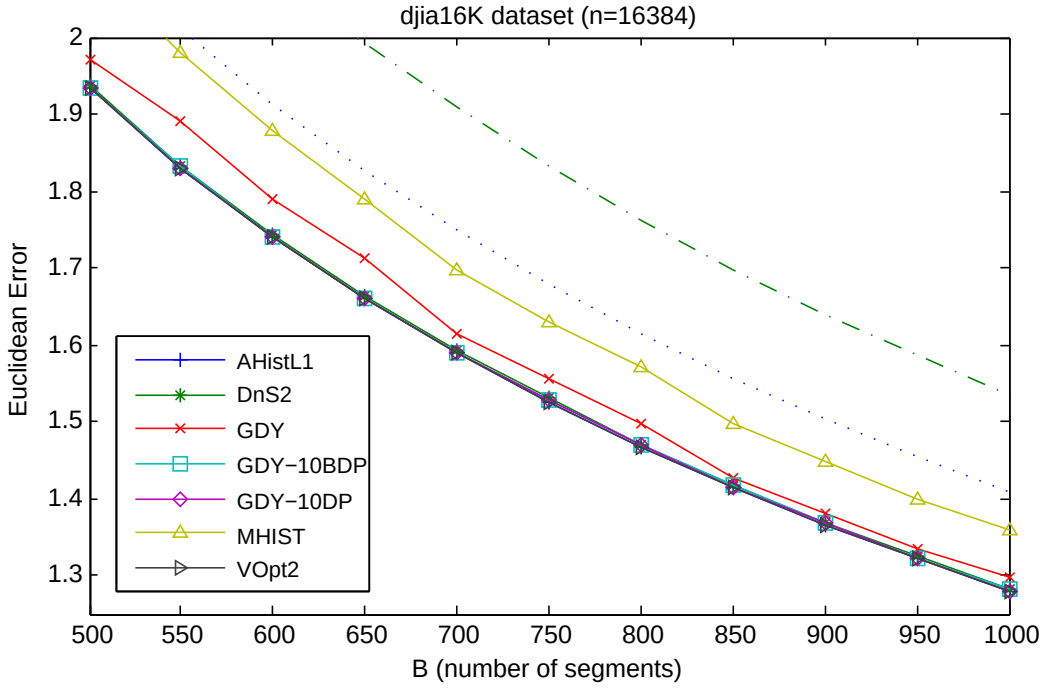


Figure 2.10: Quality comparison: DJIA

In Figure 2.10 we present the results for the *filtered* version of the DJIA data set, for a range of $B = 500 \ldots 1000$. This version, also used in [43], contains the first 16384 closing values of the Dow-Jones Industrial Average index from 1900

to 1993; a few negative values were removed. The performance evaluation with this data set follows the same pattern as before. In this case, neither MaxDiff nor the low-quality version of AHistL-Δ is depicted, as they are outliers. On the other hand, the MHIST heuristic performs slightly better than it did in previous cases. Still, our GDY algorithm performs even better, while the performance of both GDY_DP and GDY_BDP is almost indistinguishable from that of both high-performing approximation algorithms in this scale.

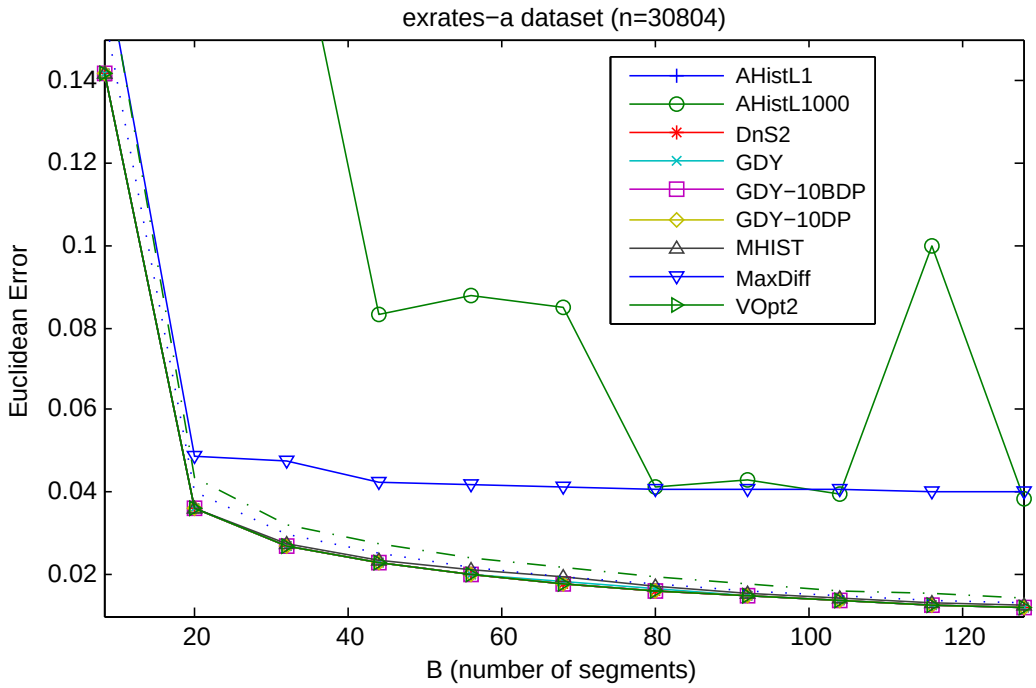


Figure 2.11: Quality comparison: Exrates

Figure 2.11 depicts the quality results with the aggregated exrates data set for $B = 8 \ldots 128$. This range of B at the lower values of the domain presents an interesting picture. Not only our advanced greedy techniques, but also the simple GDY can achieve error very close to the optimal. So does the MHIST heuristic as well, which follows at a very close distance. Still, MaxDiff and the low-quality version of AHistL-Δ remain low-performing outliers.

Next, we depict the Euclidean error results with the aggregated phone data set (Figure 2.12). The relative performance of different techniques appears clearly in this figure. GDY_DP can almost match the optimal error at least as well as the tight-ϵ variant of AHistL-Δ and DnS, while GDY_BDP and GDY follow closely behind. MHIST does not perform as well as our algorithms, while the slack-ϵ version of AHistL-Δ and MaxDiff are again poor performers.

So far we have presented quality results in terms of Euclidean error as a function of B, with the range of B zoomed in so as to allow the discernment of subtle differences. Still, we would also like to get a view of the larger picture: the

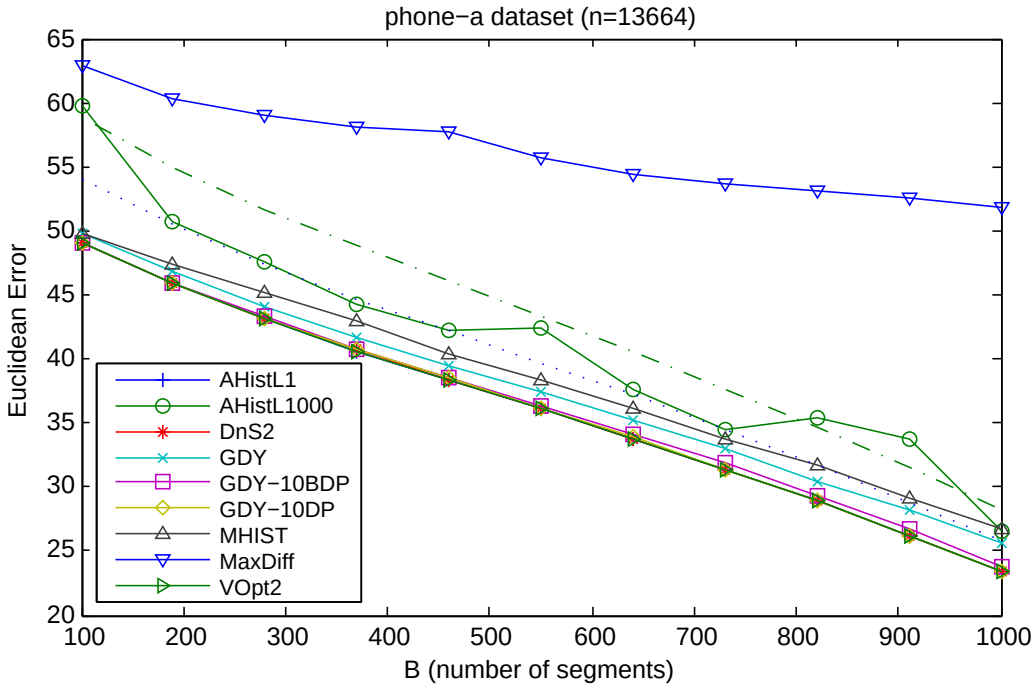


Figure 2.12: Quality comparison: Phone

shape of the $E = f(B)$ function, indicating how the Euclidean error varies over a the full domain of practical B values. We do so with the synthetic data set. This data set appears highly periodic, but never exactly repeats itself.

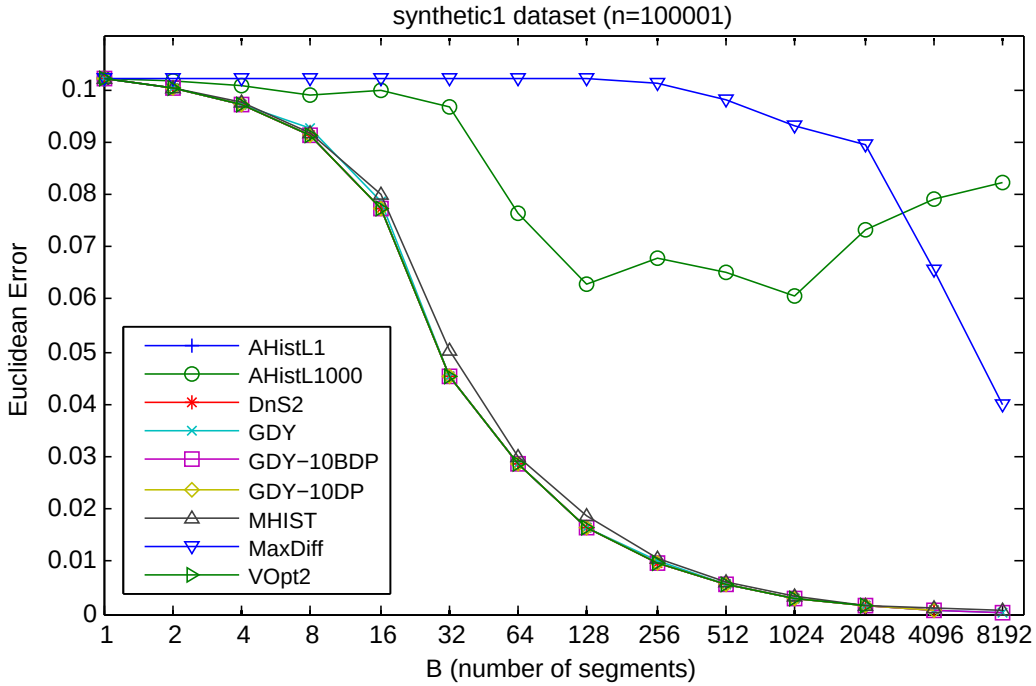


Figure 2.13: Quality comparison: Synthetic

The results are illustrated in Figure 2.13, plotting the error for a range of $B = 1 \ldots 8192$ in a *logarithmic* x-axis. These results reconfirm our previous micro-scale observation at a larger scale. The performance of the approximation

algorithms *as well as* all versions of our greedy approaches closely follow the optimal-error performance. MHIST follows them at a discernible distance. On the other hand, MaxDiff performs poorly, while the slack-ϵ version of AHistL-Δ performs unreliably. The error function is not even monotonic for the low-quality AHistL-Δ; this defect has also appeared in our earlier graphs.

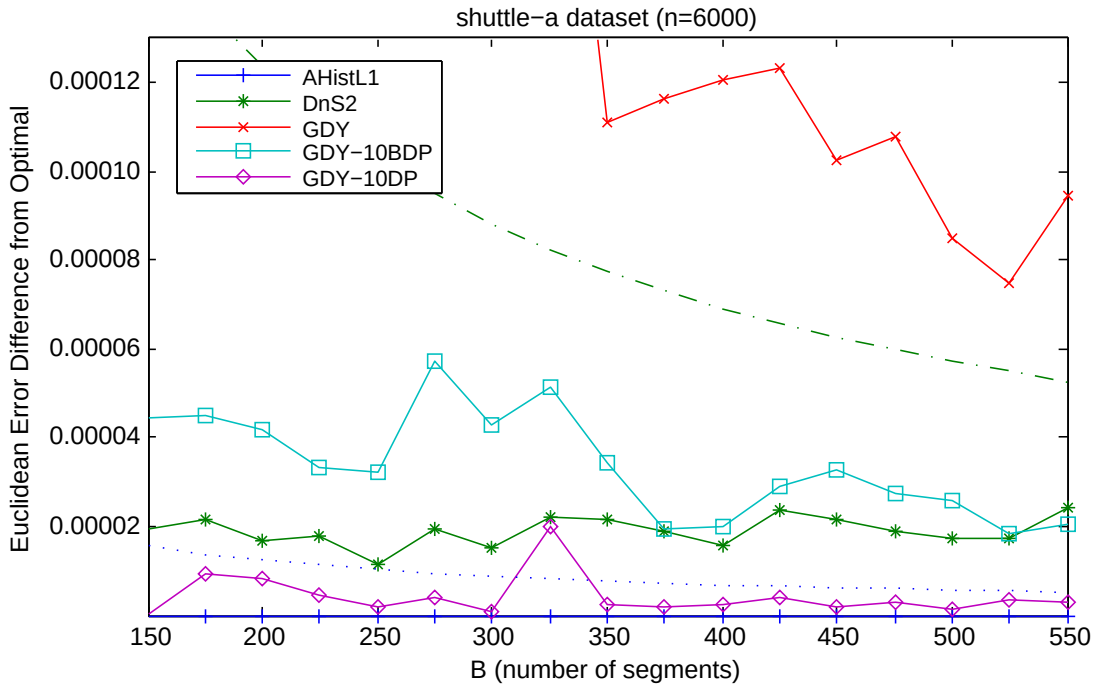


Figure 2.14: Quality comparison: Shuttle

Our results have shown the practical performance of our greedy algorithms in relation to the optimal error. Still, we would like to gain a clear view at a very close level of resolution. Thus we measure the actual *difference* of the Euclidean error achieved with the tested algorithms from the optimal error. Figure 2.14 presents the results with the shuttle data set, for $B = 150 \ldots 550$. The dotted line presents the position of 0.1% off the optimal error (achieved by V-Optimal), while the dash-dotted line traces the position of 1% off the optimal.

What was not clear before becomes apparent in this figure. AHistL-Δ matches the optimal quality, as its ϵ value predisposes it to do. The second-best performance is that GDY_DP, while DnS and GDY_BDP follow closely after. Our simple GDY algorithm does not achieve error as tightly close to the optimal, while the other heuristics are far-off, and do not fall in this figure. Still, it is remarkable that our heuristics can match and exceed the quality of DnS.

We elaborate on this line of comparison, presenting the difference from the optimal error with the aggregated winding data set (Figure 2.15). Now we use a logarithmic x-axis to present a wider range of $B = 16 \ldots 2048$. We add a dash-dash line that denotes the position of 10% off the optimal. GDY_DP exceeds the quality of DnS, while GDY_BDP comes close. GDY is more far-off, while other

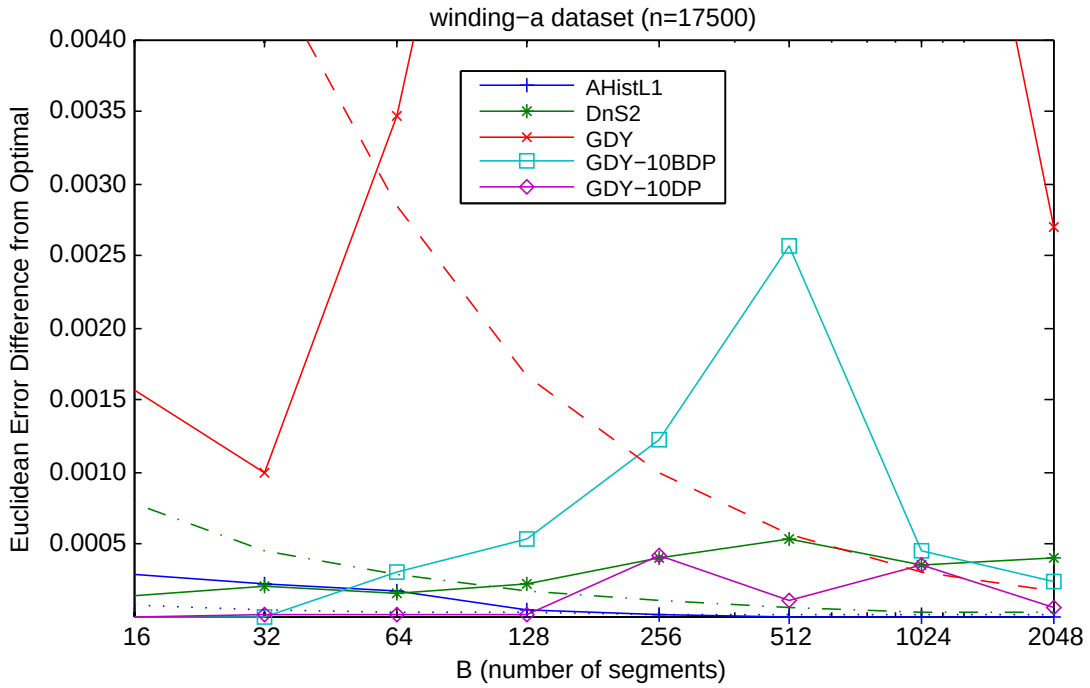


Figure 2.15: Quality comparison: Winding

heuristics are distant outliers, not seen in the figure. Thus, the injection of DP capacity into GDY indeed adds a sophistication that affords performance similar or superior to that of the guarantee-providing approximation schemes.

2.6.2 Efficiency Comparisons

Now we turn our attention to the other side of the quality-efficiency tradeoff, that of runtime performance. To get a full picture of the runtime state of affairs, we measure runtime as a function of B for constant data set size n, for varying data set size n under constant segmentation size B, as well as for B linearly varying with n.

Figure 2.16 plots the results with the *filtered* version of the DJIA data set, for $B = 500 \ldots 1000$, same as the one used for quality assessment in Figure 2.10. The time axis is logarithmic. A comparison of Figures 2.10 and 2.16 reveals some interesting findings. If the quality-efficiency tradeoff were equitably resolved by all tested techniques, then we would expect the good quality performers to be bad runtime performers, and vice versa. Still, the presented picture does not follow such a pattern. On the contrary, some of the best quality performers are also among the best runtime performers as well. That is, remarkably, our GDY_DP and GDY_BDP algorithms, which gave almost optimal quality results, also achieve satisfactorily low and scalable runtime. Moreover, GDY, which achieves next-to-optimal quality performance, is also one of the runtime champions, along with the lower-quality MHIST and the worst-quality MaxDiff heuristic. In contrast,

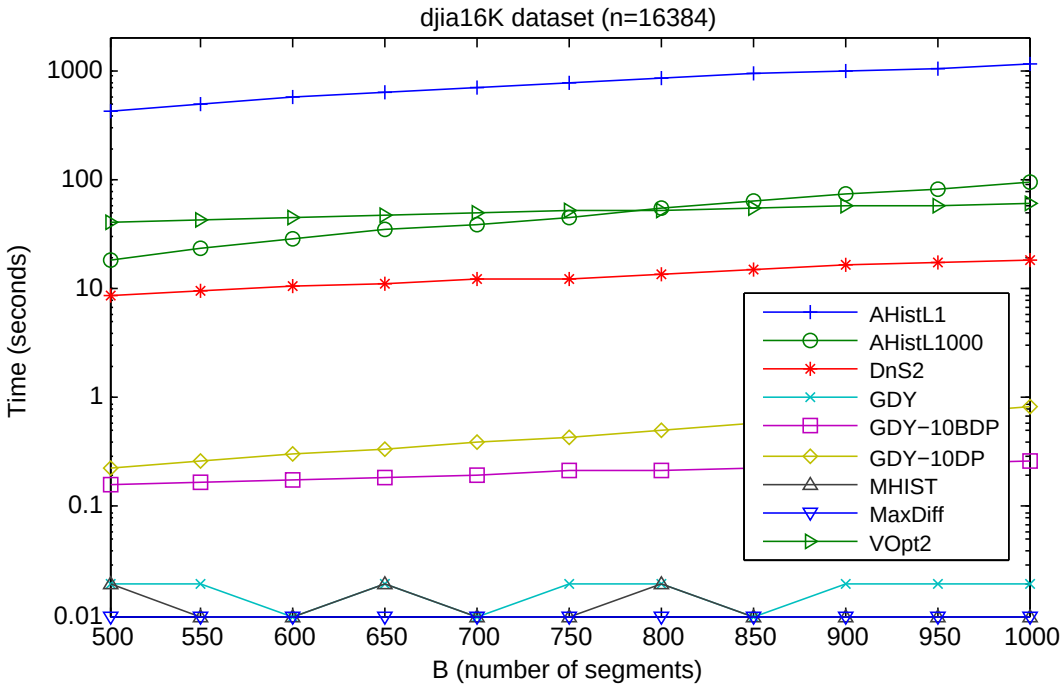


Figure 2.16: Runtime comparison vs. B: DJIA

it is clear that other high-quality performers such as AHistL-Δ and DnS pay a high runtime price for the quality they deliver. The same holds for the V-Optimal algorithm itself. Furthermore, the high-quality variant of AHistL-Δ takes runtime even higher than that of V-Optimal; the loose-ϵ variant of AHistL-Δ, which does not perform well on quality, does not gain in runtime from this looseness, and eventually exceeds the runtime of V-Optimal too. The lines in the figure indicate the cubic $O(B^3)$ complexity factor of AHistL-Δ.

Our implementation of V-Optimal, as well as of all algorithms that employ its DP scheme, follows the simple pruning step suggested in [65] (see Section 2.2). It is not clear whether the experimental evaluations in [104] and [44] have used this step. Hence, our runtime results may diverge from those reported in these works.

Next we illustrate runtimes with the aggregated winding data set (Figure 2.17), which present the runtime side of the evaluation for which Figure 2.15 shows the quality side. Axes are logarithmic, while $B = 16 \ldots 2048$. The emerging picture follows a pattern similar to the previous figure. GDY_DP and GDY_BDP achieve a remarkably attractive resolution of the quality-efficiency tradeoff, while GDY is one of the efficiency champions without sacrificing quality as MHIST and MaxDiff do. The cubic growth of AHistL-Δ is clear; both variants eventually exceed the runtime of V-Optimal (which employs pruning), while DnS also pays a high efficiency cost.

Figure 2.18 displays runtime results with the synthetic data set, i.e., the other side of the quality evaluation depicted in Figure 2.13, with $B = 1 \ldots 8192$ on

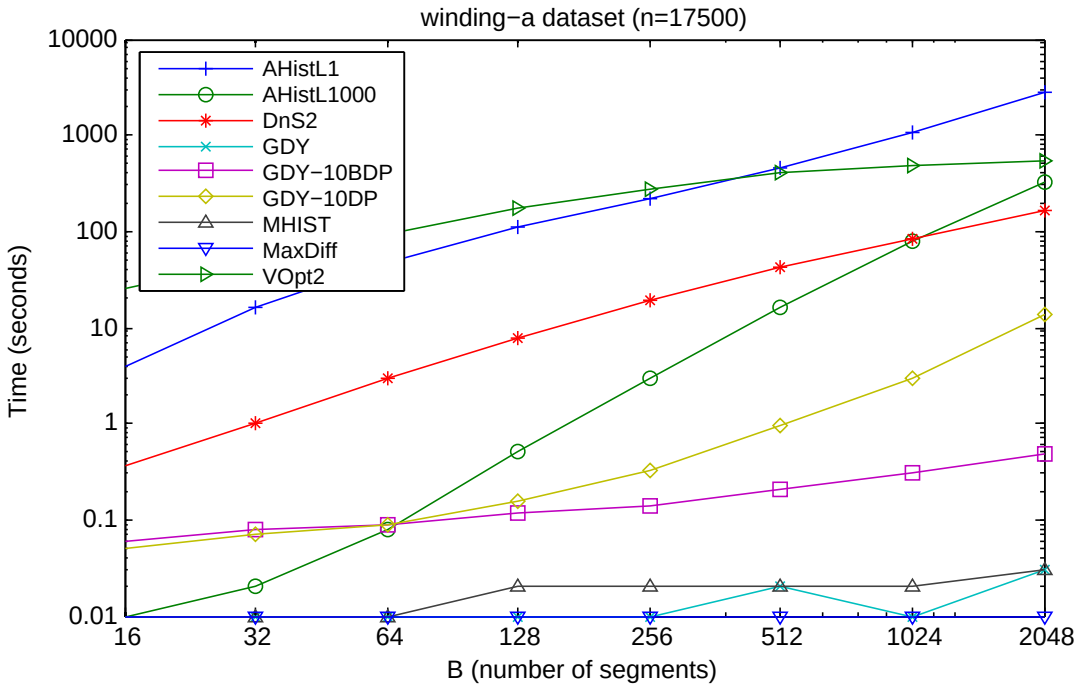


Figure 2.17: Runtime comparison vs. B: Winding

logarithmic axes. Growth trends are now more accentuated. The growth of DnS, arising from its $O(B^{5/3})$ runtime factor, is also apparent; thus, not only AHistL-Δ, but also DnS exceeds the runtime of V-Optimal for large enough B. GDY_DP also assumes an unfavorable growth trend after the pivot point of $B = \sqrt{n}$. GDY_BDP and GDY stand out as scalable algorithms that also achieve high quality.

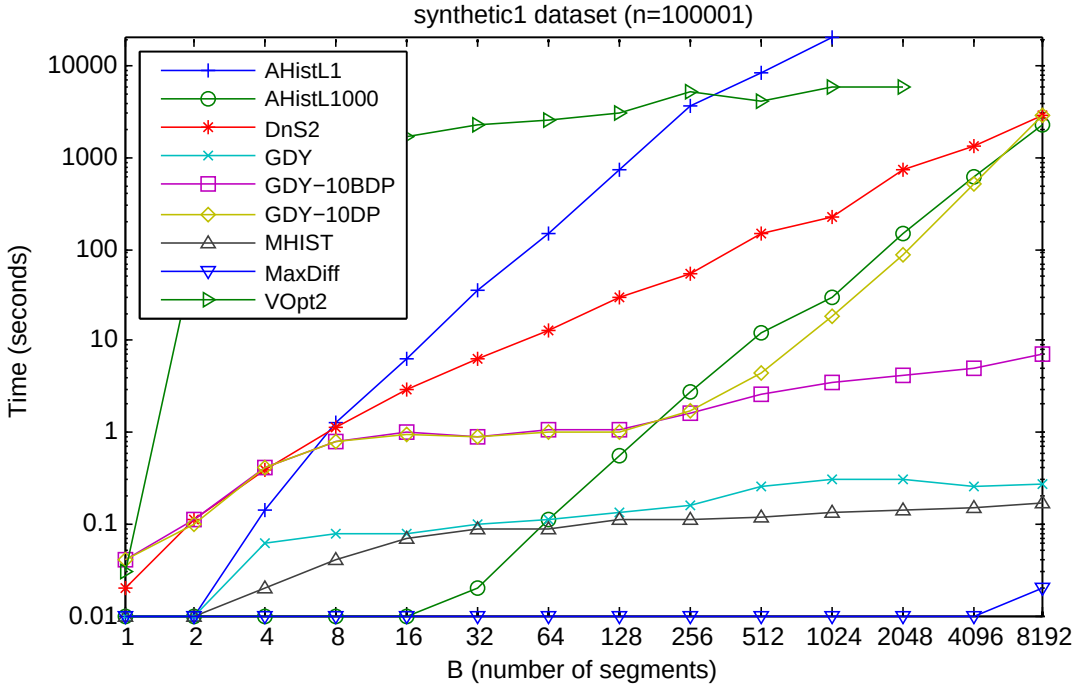


Figure 2.18: Runtime comparison vs. B: Synthetic

We also evaluate the scalability of different techniques with respect to data set

size n. Figure 2.19 outlines the runtime results for constant $B = 512$, on different prefixes of the same synthetic data set. Both axes are logarithmic. In this case, the growth of AHistL-Δ is not as severe as it was vs. B, but still stands out as the least scalable algorithm, surpassing the runtime of V-Optimal and DnS. We surmise that the $O(B^3 \log^2 n)$ worst-case complexity factor of AHistL-Δ, arising from the burden of approximating the error function itself, works out its impact more saliently as n grows. On the other hand, the impact of pruning within the DP in V-Optimal and DnS allows these algorithms to scale better, although not adequately either. In order to illustrate the impact of pruning, we also include a version of V-Optimal *without* the pruning step in this experiment (labelled VOpt as opposed to VOpt2 in the figure). The non-pruning version exhibits not only higher runtime, but also more accentuated growth with n. The champions of scalability in this experiment are again our greedy algorithms, as well as the heuristics that perform poorly on the quality side.

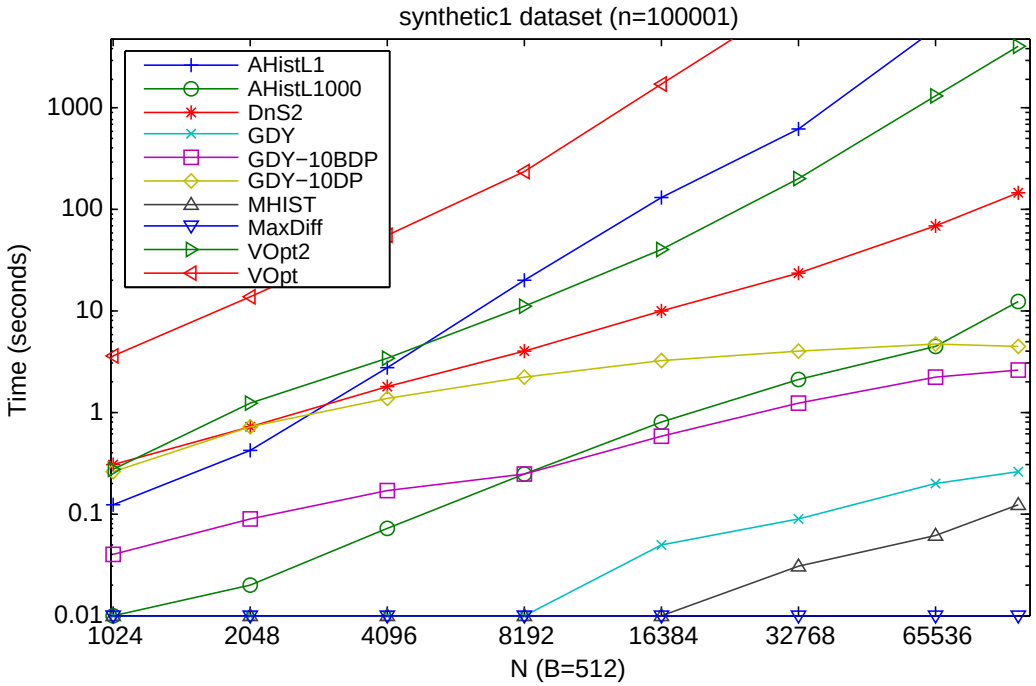


Figure 2.19: Runtime vs. n, $B = 512$: Synthetic

Finally, we measure the runtime performance with the synthetic data set when n and B grow in parallel. Figure 2.20 plots the results on logarithmic axes, where $B = \frac{n}{32}$. The unscalable growth of AHistL-Δ is conspicuous; V-Optimal and DnS do not scale well either. GDY_DP almost parallels the growth of the poorly scaling algorithms in this figure. On the other hand, the batch version, GDY_BDP, presents affordable runtime growth in this experiment too; its growth is minimally affected by the growing B, as a comparison of Figures 2.19 and 2.20 indicates, and Figure 2.18 corroborates.

Similar observations hold for GDY, MHIST and MaxDiff. Still, given their

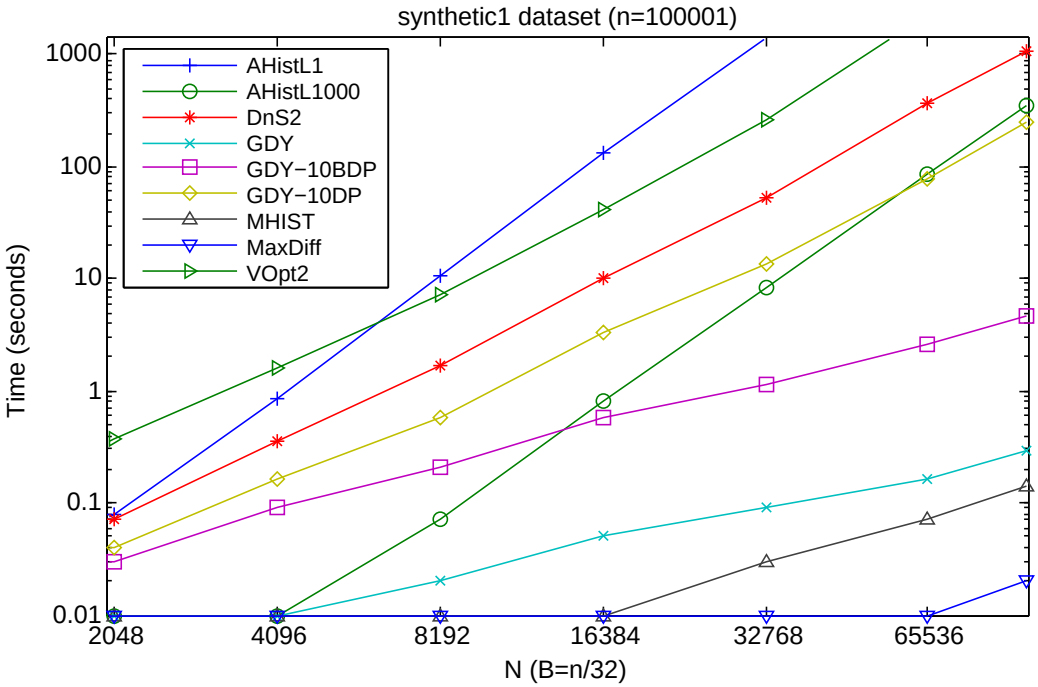


Figure 2.20: Runtime vs. n, $B = \frac{n}{32}$: Synthetic

respective quality performance, GDY_BDP emerges from our assessment as the algorithm of choice.

2.6.3 Quality vs. Efficiency Tradeoff

In the previous sections we have measured the performance of the examined algorithms in both quality and runtime, and we have inferred that some algorithms resolve this tradeoff in a more satisfactory manner than others. In particular, we have argued that algorithms like AHistL-Δ do not address this tradeoff in an attractive manner; that is, a benefit in quality comes at a high cost of runtime, while a reduction of runtime requires a severe sacrifice in quality. Still, we would like to trace this tradeoff more concretely. In this section we plot both the quality and runtime performance of examined algorithms on the same graph.

Figure 2.21 traces the quality-efficiency tradeoff with the DJIA data set and $B = 512$. The runtime axis is logarithmic, so that small runtime increases in the lower part of the y-axis are rendered noticeable. Single-version techniques are represented by a single dot. For AHistL-Δ, each variant for a different value of ϵ gets its own dot. Lower values of ϵ allow for higher accuracy at the price of extra runtime. Likewise, several variants of GDY_DP and GDY_BDP are presented, based on the number I of iterations of GDY that they use for collecting sample boundaries. More available samples enable these algorithms to achieve higher quality at the cost of efficiency. For $I = 1$ the histogram given by both these schemes is reduced to that of GDY, since one iteration produces only B samples

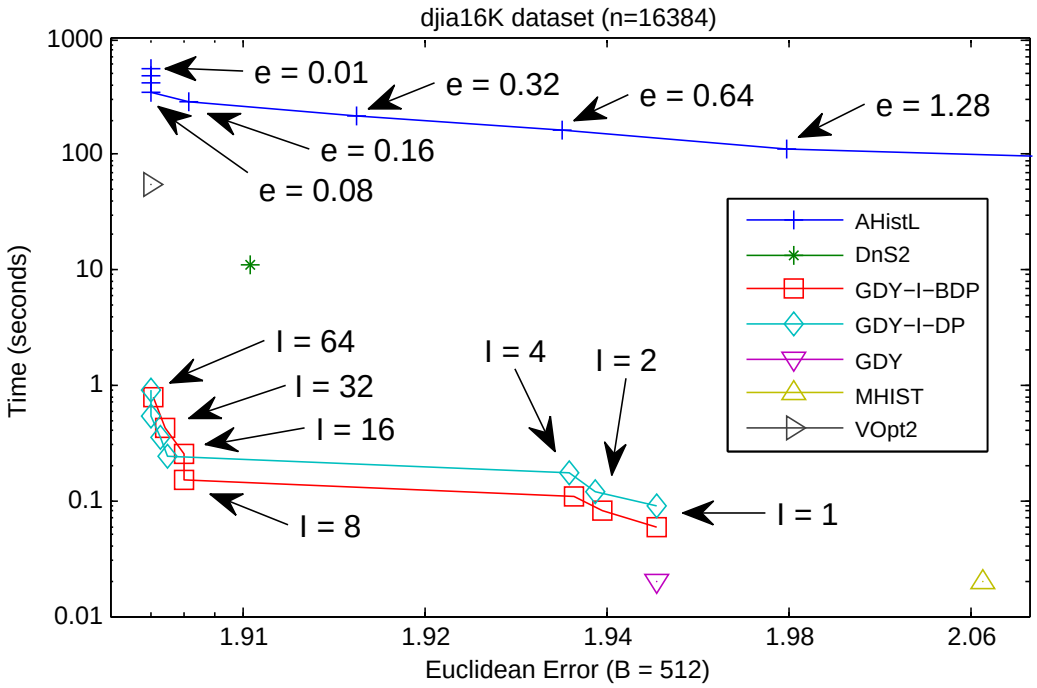


Figure 2.21: Tradeoff Delineation, $B = 512$: DJIA

to choose from.

This figure reconfirms that AHistL-Δ performs poorly at resolving the quality-efficiency tradeoff. An attempt to gain quality by lowering ϵ renders the runtime higher than that of V-Optimal; an effort to improve time-efficiency by increasing ϵ is not effective in its objective, while it deteriorates quality. DnS dominates almost all versions of AHistL-Δ in runtime and quality. In contrast, GDY_DP and GDY_BDP resolve the tradeoff attractively, while they can improve on quality by investing the extra time required by (slightly) higher values of I.

2.6.4 Local Search Sampling Effectiveness

We define a set of partitions (or candidate boundary positions) as samples. The DP can be run on the samples to produce the final segmentation of size B. V-Optimal includes all n possible partitions as samples and run DP to select B out of n partitions to produce the final segmentation which entails $O(n^2B)$ runtime complexity. As mentioned in Section 2.5.2, we can effectively reduce the number of samples from n down to $O(IB) = O(B)$ by running a (small) constant number of iterations I of the GDY algorithm where each run produces B samples. In this section, we want to measure the effectiveness of GDY as a sampling algorithm.

To measure the effectiveness of the samples, we look at the proximity of the samples in respect to the optimal set of partitions positions which produces the lowest total error, as well as the number of samples generated. We note that in all the experiments we conduct in this section, there is only one (unique)

optimal set of partitions. We can verify this by modifying the DP to show the number of unique optimal solutions. Thus, by inspecting the locations of the samples generated by several iterations of GDY in respect to the locations of the optimal partitions, we can visually measure its effectiveness. We also note that it is possible that several GDY runs produces overlapping partitions. We only collect distinct partitions positions produced by the GDY runs, thus the number of samples may be less than IB.

As explained in Section 2.3.2, DnS uses uniform sampling to collect samples from the subproblems. We compare our GDY sampling with DnS uniform sampling in terms of number of samples generated and its proximity in respect to the optimal partitions. We show the samples of the GDY on different number of iterations $I \in \{1, 2, 4, 8, 16, 32\}$. We also compare the total error of both algorithms. All the experiments in this section are using GDY_DP algorithm.

Figures 2.22 to 2.30 show the GDY sampling effectiveness on varying datasets. The components of each figure are explained as follow. The top left graph shows the number of samples vs. the number of segments. The number of samples produced by DnS is $\chi B = (n/B)^{2/3}B = n^{2/3}B^{1/3}$. The number of samples produced by GDY $I = 1$ is exactly B and for $I > 1$ is at most IB (as duplicates are removed). The vertical red dotted line denotes the $B = \sqrt{n}$. The corresponding total error ratio relative to the optimal error segmentation is given as percentage error ratio in the top right graph. The percentage error ratio is computed as $(A/B - 1) * 100\%$ where A is the total error of the algorithm and B is the optimal total error given by the V-Optimal. The next two graphs at the bottom are the proximity graphs. The proximity graph shows the proximity of the samples and the optimal partition positions. The top most oscillating blue line is the visualization of the data sequence from left to right. The red vertical lines are the positions of the (unique) optimum partitions (or the optimal segmentation). The red horizontal line lie small red circles denoting the positions of the samples produced by the DnS. The green horizontal line lie small green circles denoting the positions of the samples produced by $I = 32$ runs of GDY. We call a candidate boundary position in the samples that lies at exactly at one of the optimum partition positions a **hit** and a **miss** otherwise. We discover that a single run of GDY on the tested datasets, more than 50% of the produced samples are a hit. Running the GDY multiple times rapidly improves the percentage further. If the percentage reaches 100%, then running DP on the aggregated samples will produce the optimal segmentation. The figures show that our GDY samples partitions near or even exactly at the optimum partitions positions. A **miss** is denoted by a circle drawn slightly above the horizontal samples' line of the algorithm in the proximity graph. The number of input data sequence n,

segments B, samples, misses, the total error, as well as the percentage error ratio relative to V-Optimal of the algorithms are stated in the proximity graphs above the red and green horizontal lines. [2]

The reduction on the number of samples because of duplicate partitions can be seen in the figures by comparing the black dash-dotted lines which represent a line where the number of samples is $32 * B$ with the GDY$I = 32$ line. The **distinct** number of samples generated for GDY$I = 32$ vary as the datasets vary. On the other hand, the uniform sampling performed by DnS generates an order magnitude more samples. This contributes to the high runtime of DnS without gaining much quality improvement. In most of the tested datasets, GDY_DP with $I = 32$ produces optimal segmentations for all the range of B in test with significantly less number of samples than that of DnS.

We observe that on certain number of buckets the number of samples produced by the GDY runs are less than the other number of buckets. By observing the error ratio relative to the optimal total error, our hypothesis is that the on certain number of buckets, it is easy or obvious to the GDY algorithm to pinpoint the best positions for the partitions, yielding smaller number of samples. On cases where the data sequence is very smooth (i.e., the shuttle1 dataset), the GDY produces large number of samples since it is not clear or obvious for the best positions. Nevertheless, in such case, any single GDY run produces near optimal segmentation. Thus, our GDY can be tuned to produce less samples on such case where the samples gets very large very quick.

The effectiveness of GDY in producing samples is due to its focus in finding segmentation positions that are relevant to the data sequence. GDY correctly produce samples on the important regions of the data sequence. Due to the stochastic nature of GDY, it consistently produces good samples that within a few dozen iterations it managed to discover most if not all of the optimal segmentation positions. Our GDY effectively extracts the characteristics of the data sequence which naturally leads to better segmentation quality. In some sense, it "learns" the good segmentation positions from the data sequence itself. This technique turns out to be far more effective and efficient than existing algorithms that are not focused on the data. In contrast, AHistL-Δ focuses on the total error and discards insignificant elements in the data sequence. DnS focuses on the approximation guarantees which leads to performing uniform sampling. The heuristic approaches are deterministic and thus may not give robust segmentation quality: MaxDiff performs better on more spiked data sequence, while MHIST performs better on smoother data sequence [65].

[2] A more comprehensive visualizations on the proximity graphs are available in the website: http://felix-halim.net/histogram

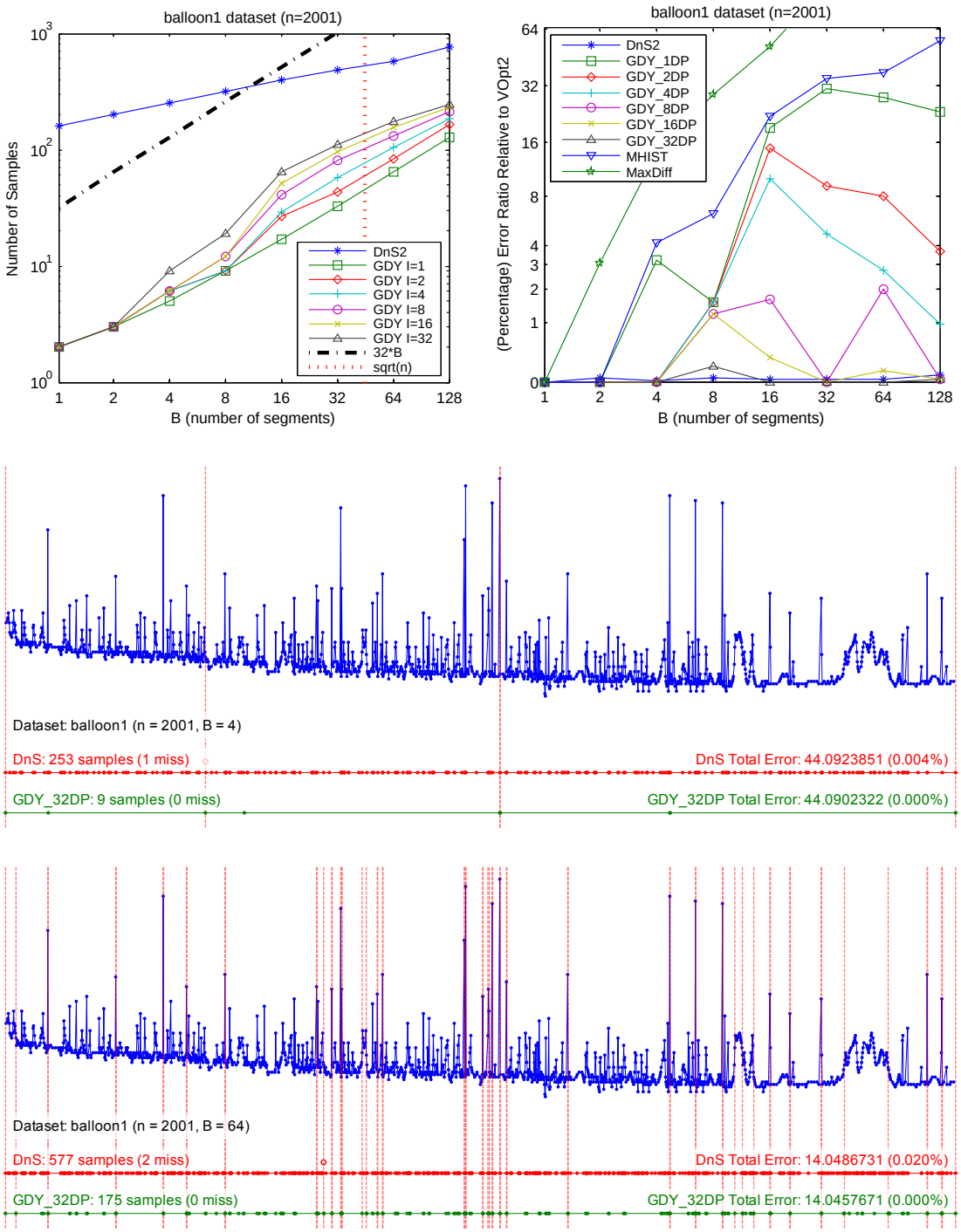


Figure 2.22: Sampling results on balloon1 dataset

Figure 2.22 shows that the number of samples produced by GDY $I = 32$ is significantly lower than $IB = 32B$. Depending on the dataset, doubling the I does not results in twice the number of samples generated. On the other hand, doubling the I significantly decreases the percentage error ratio by (often more than) twice. MaxDiff and MHIST heuristics have poor error ratio which increases as the number of segments increases. GDY_DP $I = 32$ error ratio is *on par* with that of DnS over all ranges of B in test. The proximity graph for $B = 4$ shows that the uniform sampling of DnS is not effective. It generates 253 samples and yet misses 1 optimal partition position. On the other hand, $I = 32$ runs of GDY remarkably only produce 9 samples with no miss and thus produces an optimal segmentation. Increasing the number of segments to $B = 64$ do increase the number of samples generated by the GDY. However, the proximity of the GDY samples are near that of the optimal partition positions.

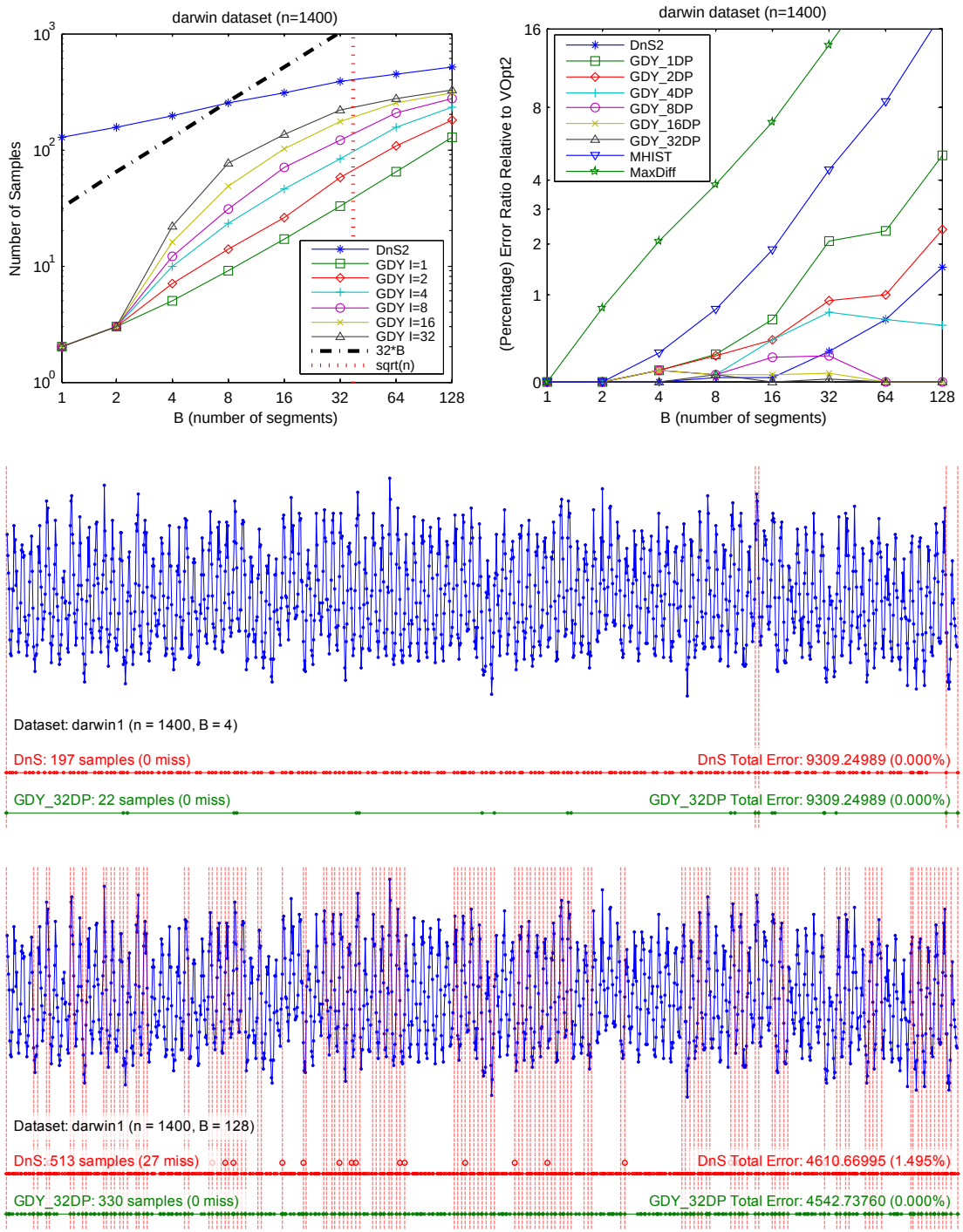


Figure 2.23: Sampling results on darwin dataset

Figure 2.23 shows that GDY $I = 32$ produces samples close to the $32B$ line which means that the GDY is having a hard time in guessing the locations of the optimal partitions. Interestingly, the error ratio graph shows that GDY_DP $I = 4$ error ratio is on par with that of DnS and GDY_DP $I = 32$ has perfect error ratio of zero across all ranges of B in test. This result suggests that the sampling quality of our GDY outperform that of DnS on hard dataset like darwin. The proximity graph shows that both GDY and DnS roughly uniformly samples across the entire sequence. However, the number of misses for DnS is far higher.

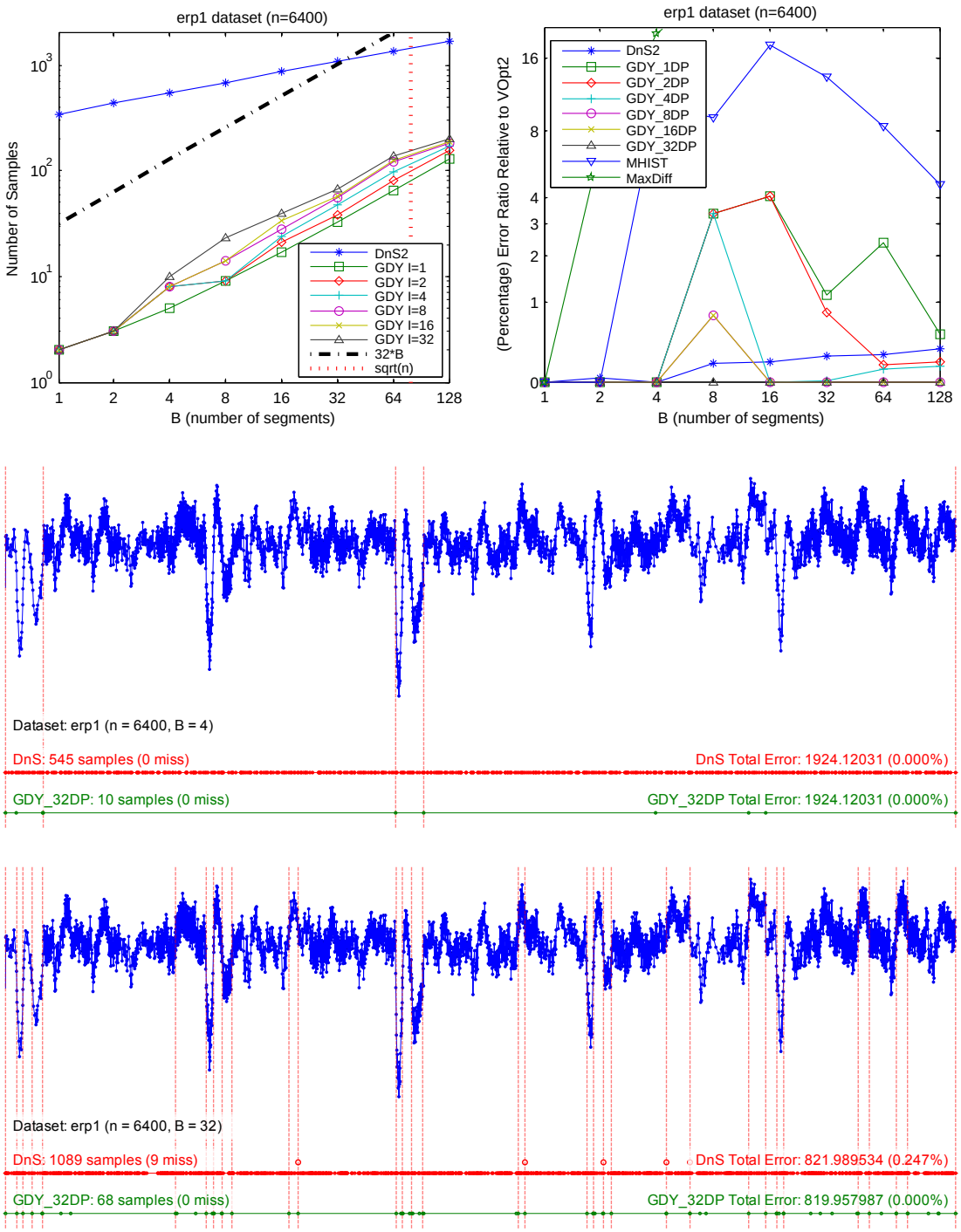


Figure 2.24: Sampling results on erp1 dataset

Figure 2.24 shows that erp1 is an *easy* dataset for GDY. Over all ranges of B in test, the number of samples for GDY is relatively small. That is, multiple iterations of GDY produces many overlapping partitions (duplicates are removed). The error ratio of GDY_DP $I = 8$ is on par with that of DnS while GDY_DP $I = 32$ always give optimal segmentations. GDY $I = 32$ generates significantly less samples than that of DnS. The error ratio for MaxDiff and MHIST are significantly higher than a single run of GDY. The proximity graph shows the high quality of GDY sampling that the 68 samples of $I = 32$ runs of GDY reside close to the optimal partition positions without any miss.

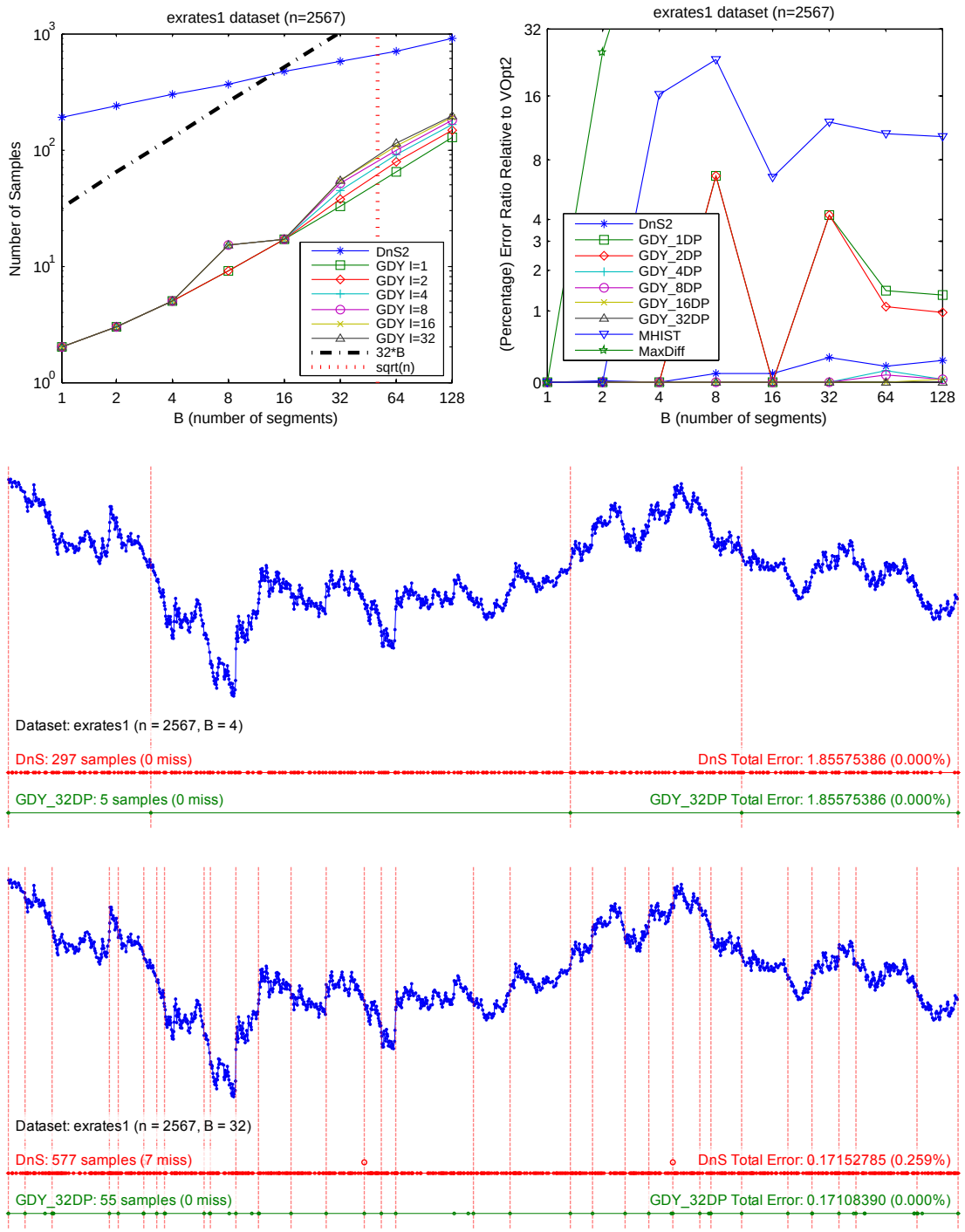


Figure 2.25: Sampling results on exrates1 dataset

Figure 2.25 shows similar results with Figure 2.24 that exrates1 is another friendly dataset for GDY. We consistently see that for very small value for $B = 4$, GDY has no difficulty in finding near optimal or optimal partitions positions.

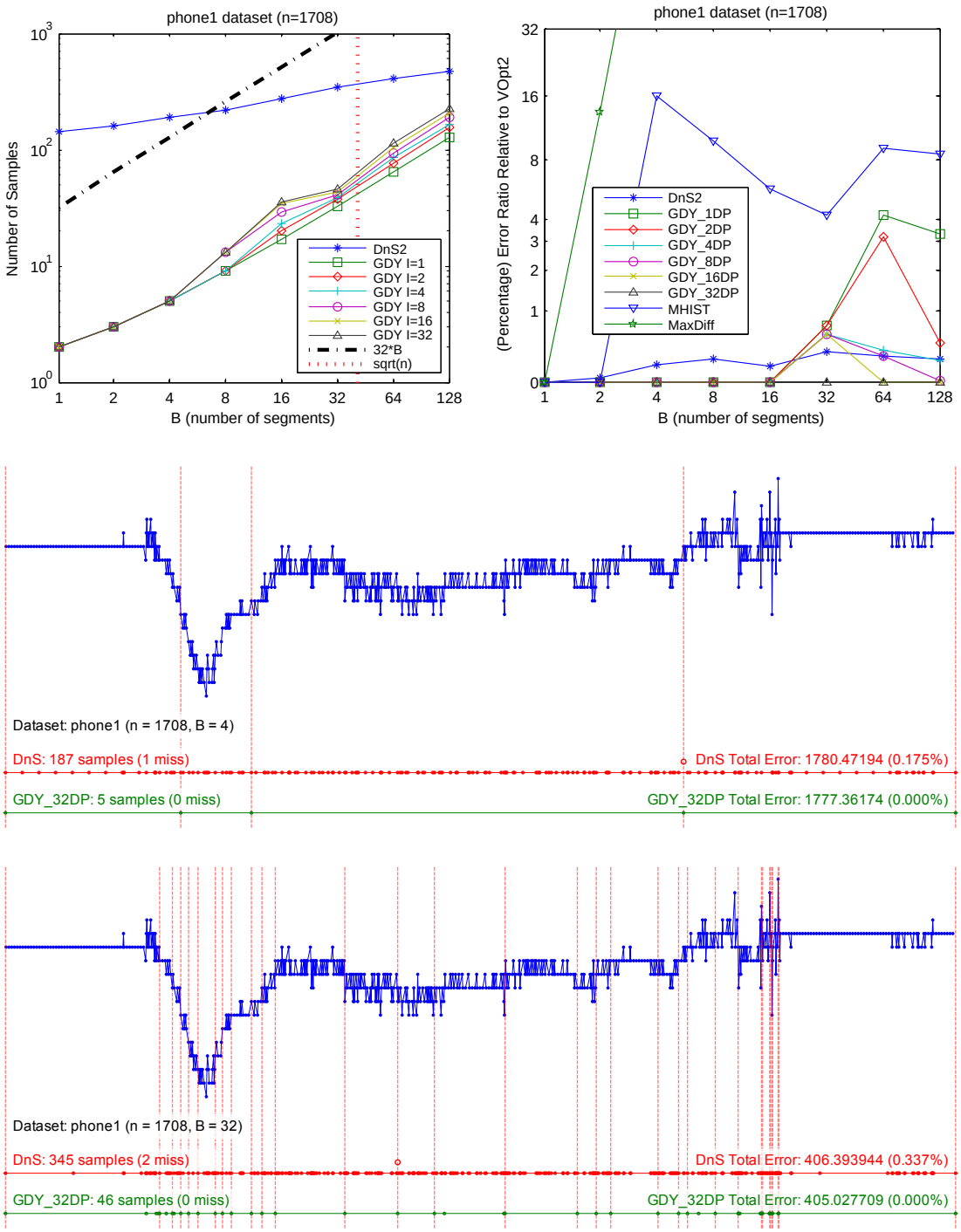


Figure 2.26: Sampling results on phone1 dataset

Figure 2.26 shows the (surprising) power of GDY. A single run of GDY manages to produce optimal segmentation, thus it outperforms the other algorithms for small $B \in 2, 4, 8, 16$. For $B >= 32$, $I = 4$ is on par with DnS. Again, the proximity graph shows the precision and efficiency of the GDY samples.

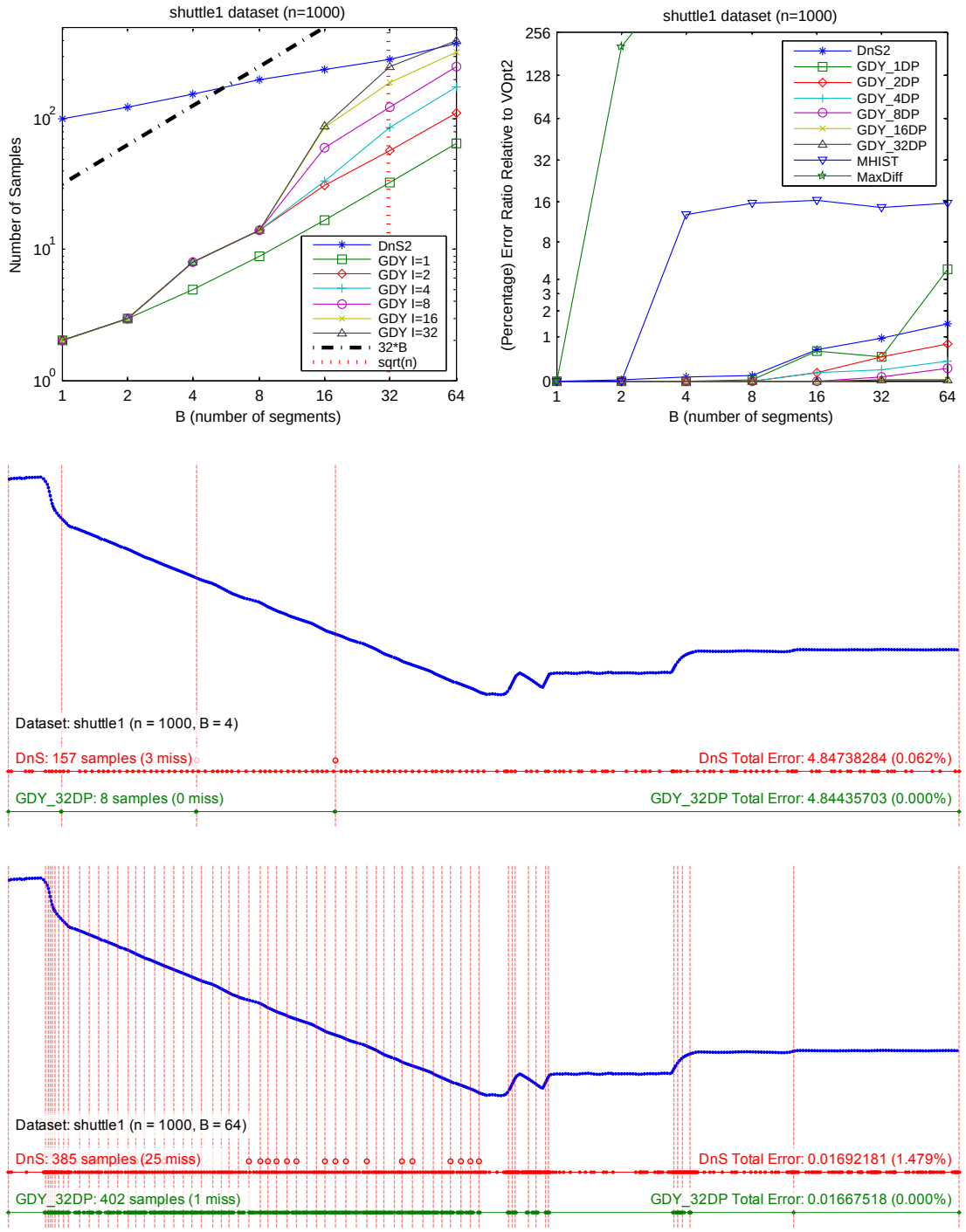


Figure 2.27: Sampling results on shuttle1 dataset

Figure 2.27 shows that shuttle1 is a tricky dataset for GDY. A smooth progression data sequence can explode the number of samples generated by GDY. We see that for $B >= 16$, the number of samples grow large very quick and exceeds the number of samples that of DnS for $B = 64$. The proximity graph shows that many of the GDY samples are spent on the smooth progression of data sequence. The shuttle1 is another hard dataset besides the darwin dataset. However, both cases shows that if the number of samples becomes very large very quick, it is reasonable to decrease the number of iterations I. In the darwin dataset (see Figure 2.23), with $I = 4$ already give a good segmentation with similar error ratio with that of DnS. In the shuttle1 dataset (see Figure 2.27), with $I = 2$, it outperform DnS over all ranges of B in test. Our hypothesis is that if the dataset is hard (i.e., GDY sampling degenerates to uniform sampling), the number of samples generated by GDY will get large very quick, however, the error ratio will be already small enough. Thus, we can prematurely stop the sampling process once the number of samples exceeds some threshold and yet we still have a fairly good segmentation quality.

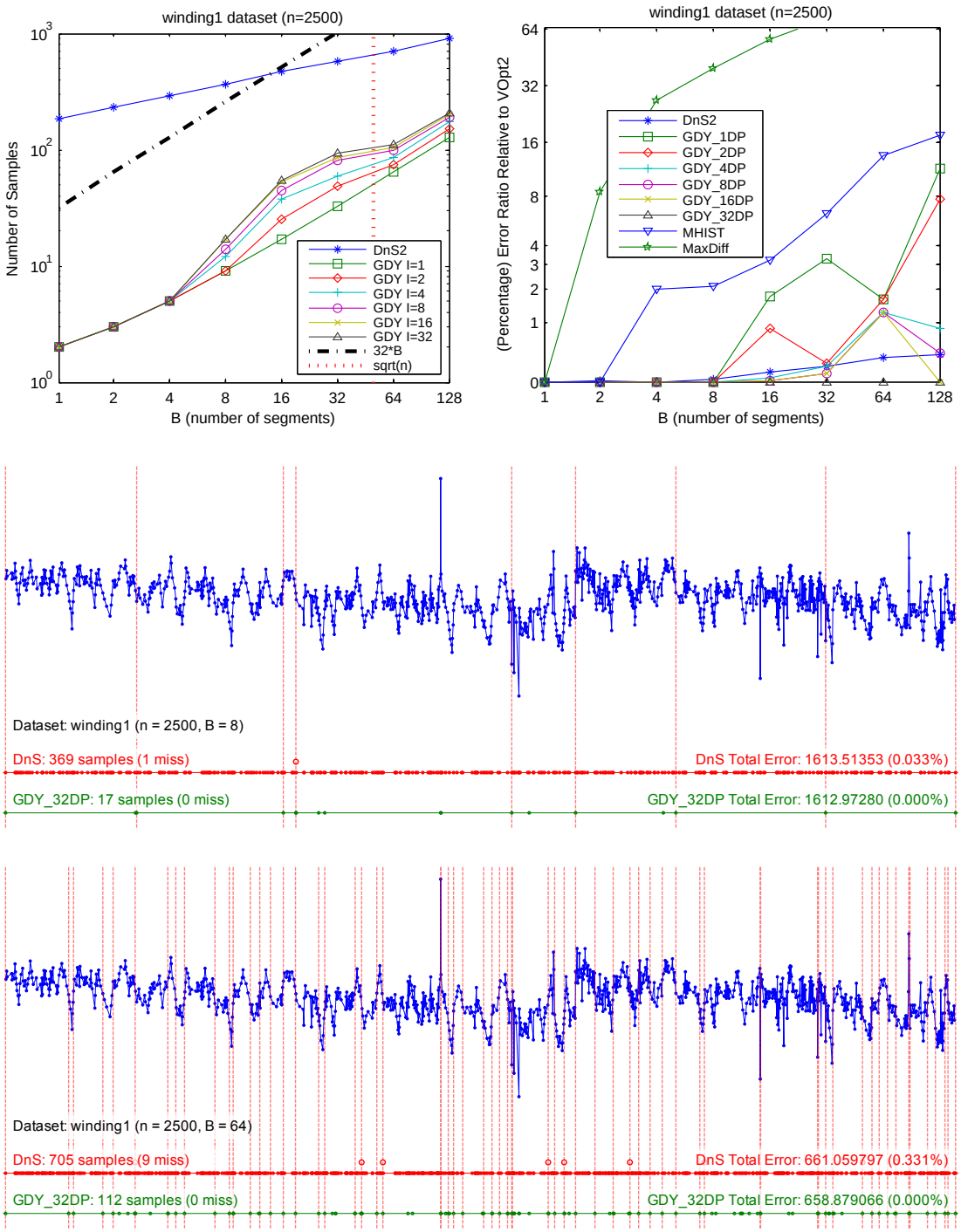


Figure 2.28: Sampling results on winding1 dataset

Figure 2.28 shows similar results with Figure 2.23 on the darwin dataset. The number of samples grows in the $B \in [8, 64]$ range, and the error ratio gets bigger as B gets higher. On the contrary, in the proximity graph for $B = 8$, MaxDiff that is supposed to be better on spiked data perform poorly in this dataset. Unfortunately for MaxDiff, for some number of segments (i.e., $B = 8$), the optimal partition positions do not even nearly lie on the highest difference between two consecutive values in the data sequence.

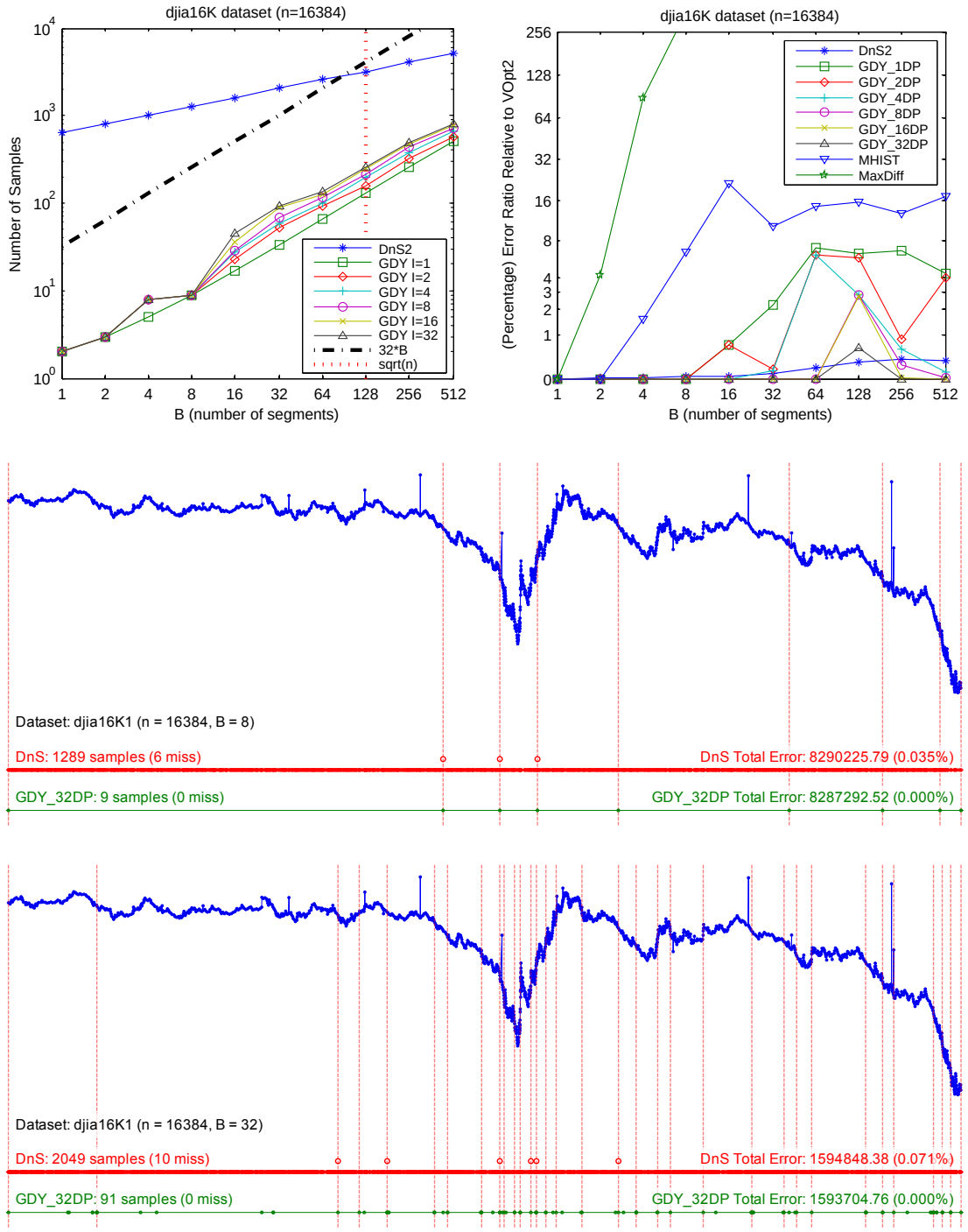


Figure 2.29: Sampling results on djia16K dataset

Figure 2.29 shows a larger data sequence with $n = 16384$. With bigger dataset size, DnS requires more samples since the number of samples produced by DnS is dependent on both n and B, which is $n^{2/3}B^{1/3}$. For $B = 8$ we see that DnS produces more than two order magnitude more samples than GDY $I = 32$ and yet they produce the same error ratio. MaxDiff gets even worse as the dataset and the number of segments gets larger. As shown in the proximity graphs, GDY maintains its conciseness in sampling large data sequence without any miss, as if it already knows where the optimal partitions positions reside.

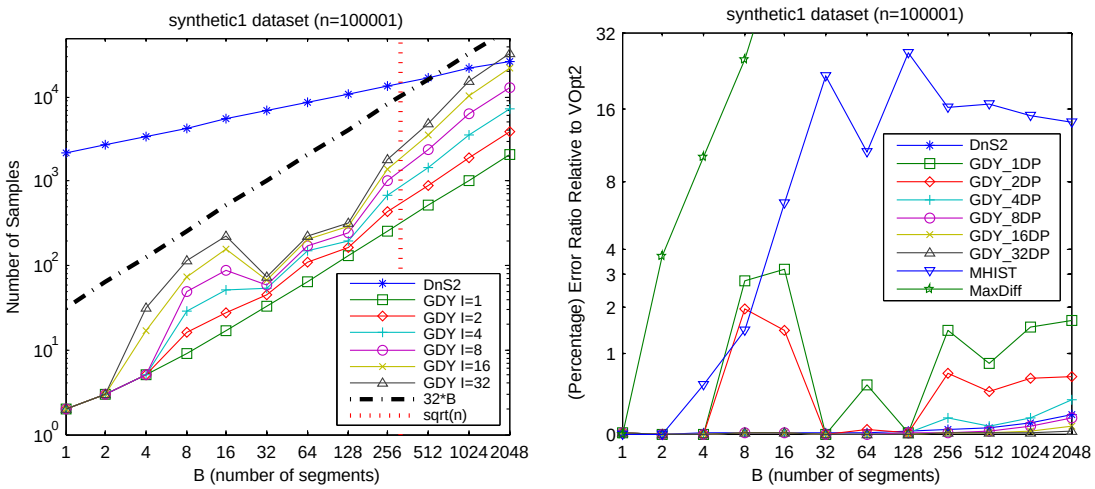


Figure 2.30: Sampling results on synthetic1 dataset

Figure 2.30 shows the result of an even larger data sequence $n = 100,001$. Interesting phenomenon occurs on $B \in [32, 128]$. The number of samples decreases significantly on those ranges. We could not provide the proximity graphs for the synthetic1 dataset due to its size. However, we conjecture that the decrease of number of samples for $B \in [32, 128]$ is due to the easiness/hardness part of the dataset. The error ratio graph shows that for $B \in [32, 128]$, the error ratio is very small. Which suggest it is easy for GDY to locate the optimal partition positions on those ranges. GDY_DP with $I = 4$ is on par with DnS. MaxDiff and MHIST performs very poor in this dataset.

2.6.5 Segmenting Larger Data Sequences

To demonstrate the scalability of the algorithms, we combine the 10 Synthetic datasets into a longer data sequence of length $n = 1,000,010$. In this experiment we only have a time resources up to 10,000 seconds thus algorithms that need more than 10,000 seconds are considered impractical. We note that most mobile applications have far significantly lower resources. Figure 2.31 shows the runtime performance of the algorithms. The approximation algorithm AHistL $-\Delta$ and DnS practically work only for B less than several hundreds. GDY performance is relatively better than MHIST for large enough B. While MaxDiff achieves the fastest runtime, it produces the worst segmentation quality. GDY_DP as expected is only practical for $B \leq \sqrt{n}$ while GDY_BDP is scalable for very large B.

To compare the quality of the segmentation among the algorithms, we use the segmentation produced by the GDY_10BDP algorithm as the base comparison. Figure 2.32 shows the relative error difference to the GDY_10BDP. AHistL-Δ, DnS, GDY_10DP give a very close total error to GDY_10BDP on all practically runnable values for B. GDY and MHIST give poor quality on $B \in [32, 4096]$, however as B gets very large, both starts to give better quality. This supports our intuition (mentioned in Section 2.5.3) that the larger the B, the less the significance of the

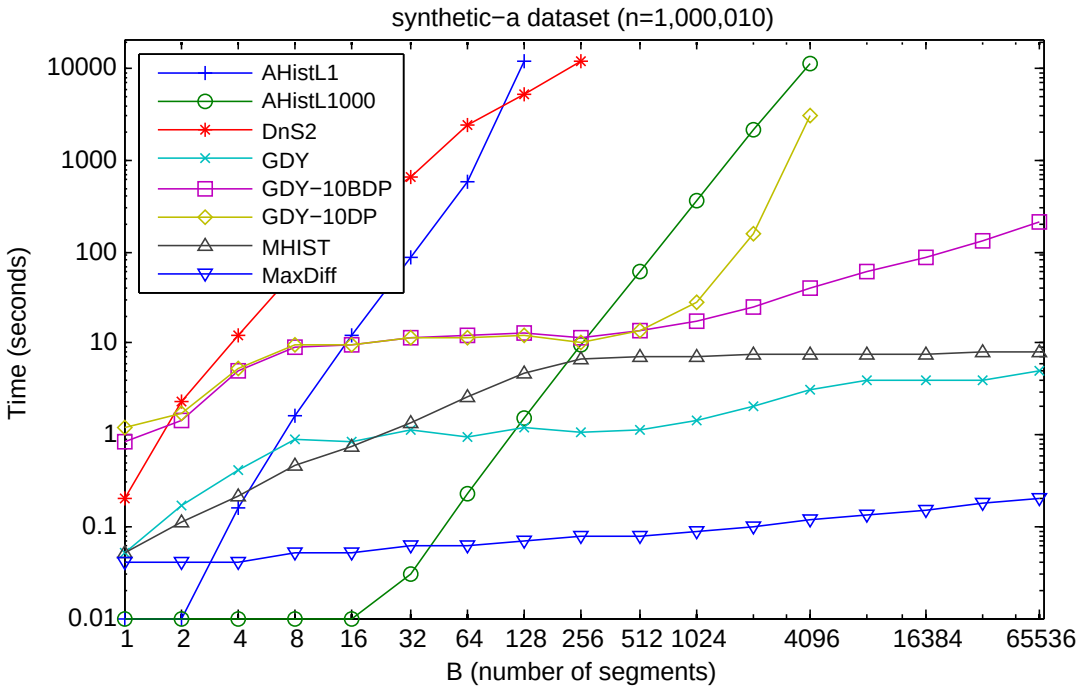


Figure 2.31: Number of Samples Generated

individual samples in its contribution to the total error. MaxDiff is an outlier (it has far bigger error than MHIST) and not shown in the graph.

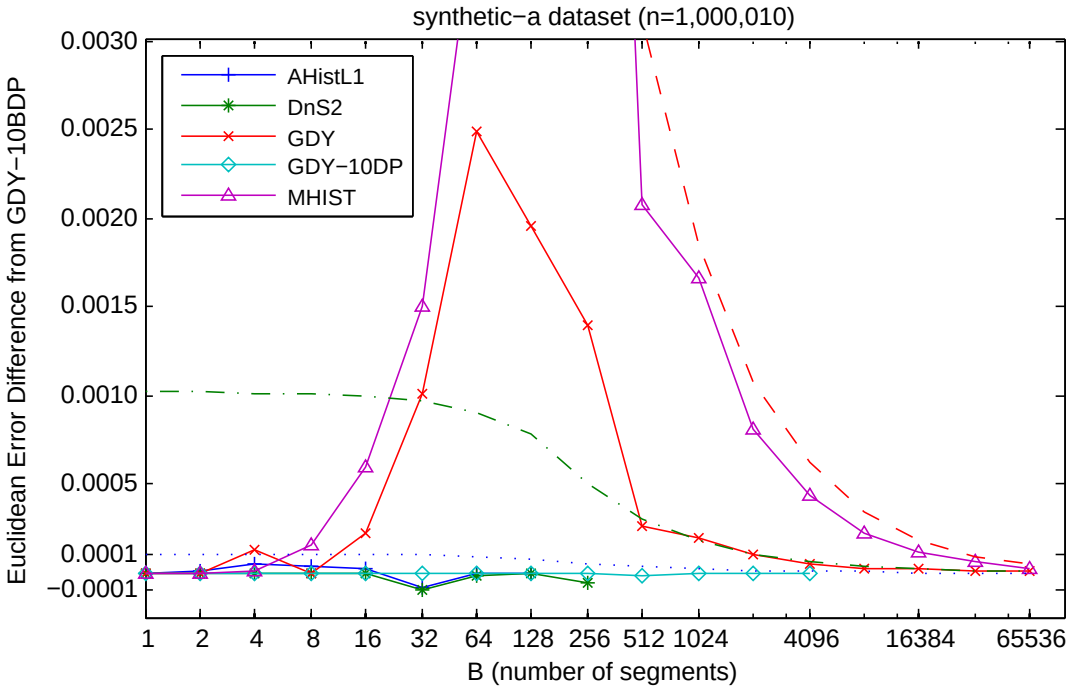


Figure 2.32: Relative Total Error to GDY_10BDP

We give the same tradeoff graph as in Figure 2.21 for the combined synthetic-a dataset. Figure 2.33 shows the tradeoff for $B = 64$ while Figure 2.34 shows the tradeoff for $B = 4096$.

For $B = 64$, we see that GDY_DP and GDY_BDP has similar quality as well as

performance. DnS on the other hand shows worse performance than AHistL-Δ. AHistL-Δ still does not give satisfactory tradeoff for different value of ϵ. MHIST gives fast runtime but poor quality.

For $B = 4096$, we are not able to practically run the DnS and AHistL-Δ algorithm. With large B, we see that GDY_DP performance starts to drop. As expected GDY_BDP manages to keep good performance while maintaining the segmentation quality over different I. Nevertheless, GDY_BDP requires more iteration ($I = 32$) to achieve the segmentation quality of GDY_DP (with $I = 8$). Finally, GDY has both better runtime and quality compared to MHIST.

Overall, GDY_BDP gives the best compromise on quality and performance in processing a very large data sequence.

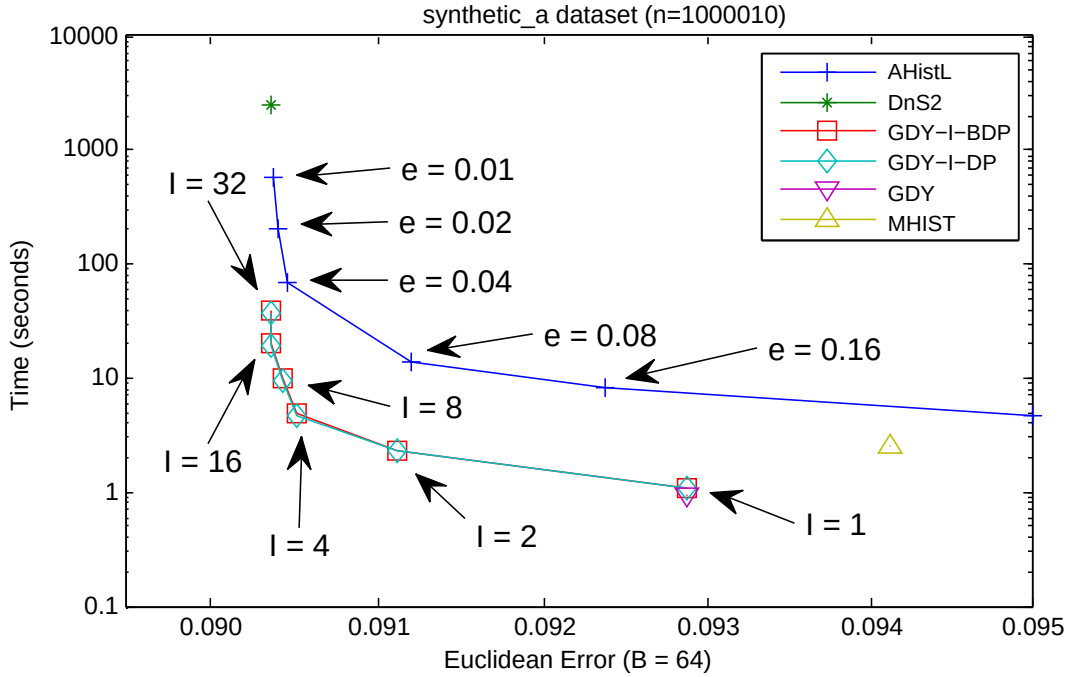


Figure 2.33: Tradeoff Delineation, $B = 64$

2.6.6 Visualization of the Search

In the foregoing study, we have assessed the degree to which a given segmentation $\mathbf{S}$ approximates the optimal one $\mathbf{S}^*$ in terms of the Euclidean error metric. Another way of assessing the convergence towards the optimal segmentation is to count the amount of bucket boundary positions that $\mathbf{S}$ and $\mathbf{S}^*$ do *not* have in common, or its *distance* to $\mathbf{S}^*$. The optimal solution is achieved when the distance is 0. The advantage of this distance metric is that it conveys an intuitive representation of how far a given segmentation is from the optimal one, in a way that an error value does not.

In this Section, we present our evaluation of the algorithms we study in terms

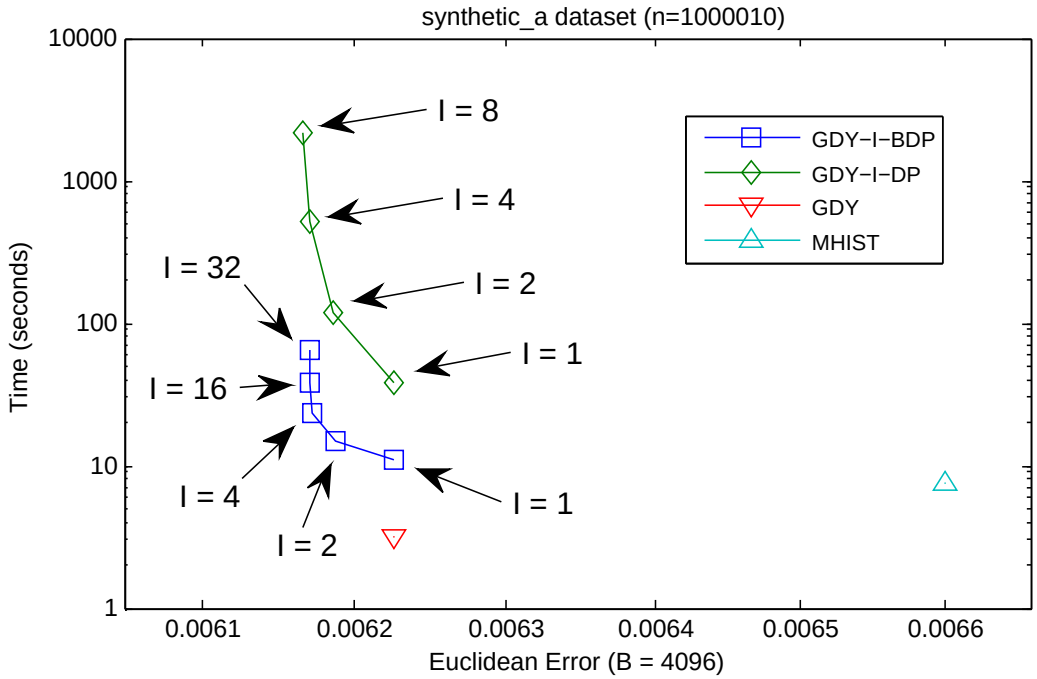


Figure 2.34: Tradeoff Delineation, $B = 4096$

of the distance metric. Let a segmentation $\mathbf{S}$ be expressed as the set of B boundary positions that define it. Then *distance* between two segmentations $\mathbf{S}_1$ and $\mathbf{S}_2$ is defined as $d = B - |\mathbf{S}_1 \cap \mathbf{S}_2|$. A distance value d indicates that $\mathbf{S}_1$ has d boundaries that do not match any boundary in $\mathbf{S}_2$, and vice versa.

[50, 51] proposed a novel visualization, *Fitness Landscape Search Trajectory* (FLST), for visualizing the behavior of local search. We have adopted this technique to visualize the progress of GDY_DP as it collects samples from GDY runs. Figure 2.35 shows a two dimensional visualization which shows how different solutions from the different algorithms compare in terms of the distance to $\mathbf{S}^*$ as well as pair-wise distance among them. The visualization is intended to give an approximation to the distance between solutions – solutions with small d are close, while those with a larger d value are further apart. The indicated value of d is the distance of that solution compared to $\mathbf{S}^*$ (Vopt2) which is in the middle. The error ratio percentage from the optimal Euclidean error for each algorithm is also shown. The visualization depicts: two GDY_DP runs, for $I = 2$ and $I = 8$; eight different GDY runs; AHistL-Δ with $\varepsilon = 0.16$; and DnS. The runtime of each algorithm is presented in the lower graph which is aligned with the upper diagram and the time axis uses a logarithmic scale.

With respect to $\mathbf{S}^*$, it is noteworthy that each of the 8 GDY runs achieves distance of around 58-74 out of a maximum possible value of 512. In other words, a single run of GDY manages to discover ∼85% positions that belong to the optimal segmentation positions. This is a fairly good result, comparable to that of

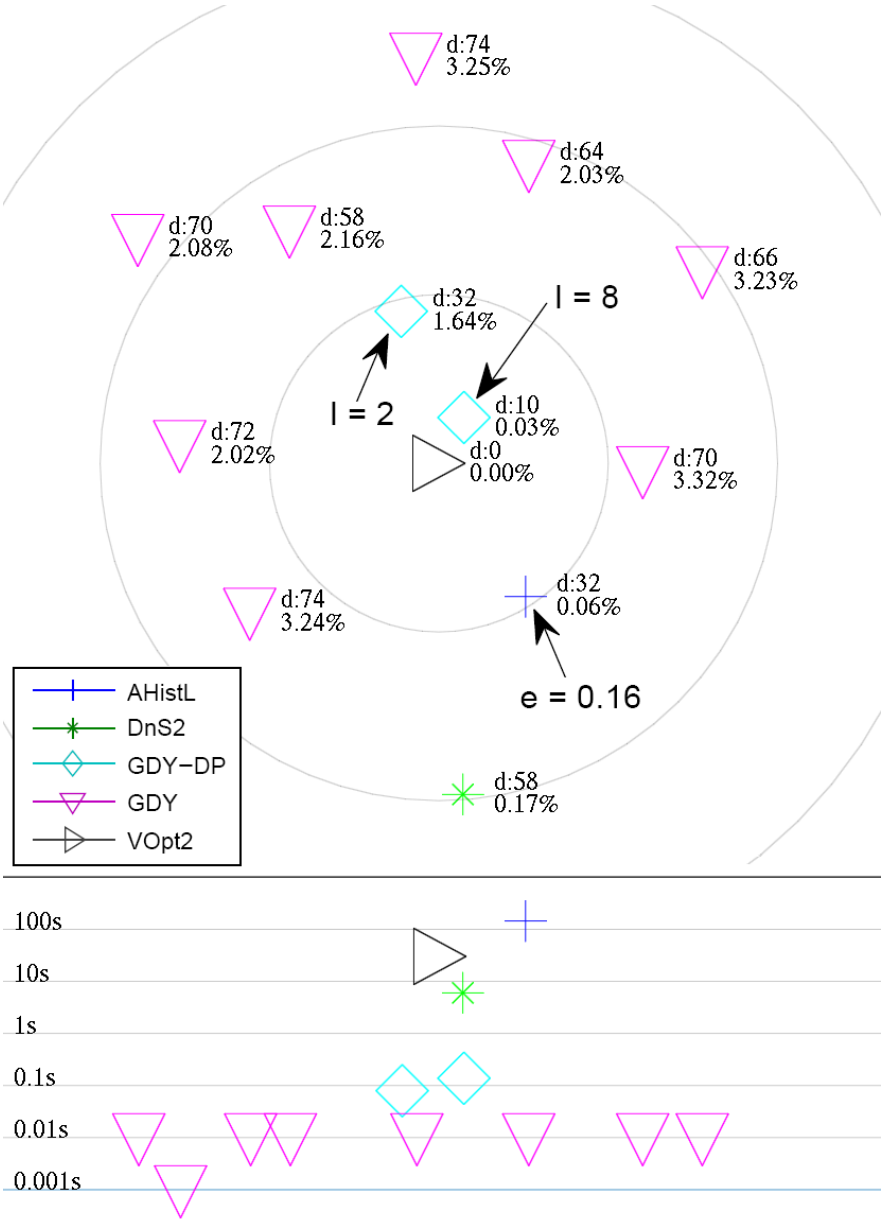


Figure 2.35: Comparing solution structure with quality and time, $B = 512$: DJIA

DnS. By exploiting only two different GDY runs, GDY_DP with $I = 2$ manages to decrease the distance to 32 (discovered ∼94% optimal segmentation positions), which is similar to the distance achieved by AHistL-Δ. GDY_DP with $I = 8$ moves much closer to $\mathbf{S}^*$ with minimal runtime increase and found ∼98% optimal segmentation positions ($d = 10$). When $I = 32$, the 32 GDY runs discovers all optimal segmentation positions and thus GDY_DP achieves the exact optimal solution with runtime less than 1 second.

The visualization shows that the solutions from different GDY are distributed around the graph which indicates that the GDY runs constitute a diverse set centered around $\mathbf{S}^*$. This diversity is needed in order for GDY_DP to be able

to take advantage of a multitude of sample boundaries. As we have discussed, the diversity is not solutions being different due to randomness. Rather all GDY solutions, while different from each other, are valid solutions which are relatively close to the optimal one.

The visualization also shows that the approximation algorithms (AHistL-Δ and DnS) do not optimize on the distance as in our GDY algorithm. Both AHistL-Δ and DnS have large distance (32 and 58) to the **S** while having very low error (0.06% and 0.17% off from V-Optimal). This shows that the approximation algorithms are not focusing on the segmentation positions that are relevant to the data sequence. Thus they do not take advantage from the characteristics of the underlying data sequence.

2.7 Discussion

Our study has led to some findings not noticed by the numerous literatures in segmentation. First, despite their elegance, approximation schemes with robust error guarantees do not achieve an attractive resolution of the tradeoff between efficiency and quality. Among the proposed schemes, DnS achieves a slightly better tradeoff than AHistL-Δ. Worse still, both can be superseded in time efficiency by V-Optimal, which they are meant to approximate, due to their super-linear dependence on B. Secondly, by employing a local search that exploits the optimal dynamic programming segmentation and sampling, we address the tradeoff much more satisfactorily. Our best performing algorithm, GDY_BDP, consistently achieves near-optimal quality in linear time.

In developing a good local search algorithm, one needs to be creative in designing heuristics to guide the local moves. Such local search tuning is known to be tedious and very time consuming and often requires an involved experimentation and evaluation process. Furthermore, the approaches taken are customized to a particular instance of the problem, which makes it hard to generalize to other unrelated domains.

We present an alternative improvement strategy for problems where there is an efficient (and in our case, optimal) algorithm for a sub-problem. In this case, local search is used to collect a diversified set of solutions. One has to devise the local search to give sufficiently good quality (which GDY achieves). The efficient improvement algorithm can improve part of the solution using the collected solutions from local search. Thus, we show that local search can be effective not just on intractable problems but also in polynomially solvable problems that present a premium in terms of efficiency. Such problems arise frequently in the fields of data mining and data engineering. We show that the "unreasonable effectiveness

of local search" also succeeds here. Furthermore, any improvements/tuning/new heuristics in the local search would likely automatically translate to improvements in the overall algorithm as it could be employed to get better samples or run more efficiently.

Our solution employs a novel local search which maintains a population of solutions and uses a recombination of those solutions. It also exploits careful use of an optimal polynomial time optimization procedure. We think that the strategy used here may also be applicable to other problems with pseudo-polynomial optimization algorithms. Such a strategy may allow for effective scalable algorithms that exploit a synergy between local search and an optimal, but more expensive, pseudo-polynomial time algorithm.

To understand the difference between our approach and more typical local search approaches, we compare GDY_DP against a pure stochastic local search version, GDY_LS, which runs GDY runs for I iterations starting with a random initial partition. The difference between the two is that GDY_LS selects the best solution found within some iteration, while GDY_DP recombines solutions found in the previous iterations using V-Optimal. Our experiments use the same random seed, hence the i^{th} iteration in both GDY_LS and GDY_DP produce exactly the same i^{th} solution.

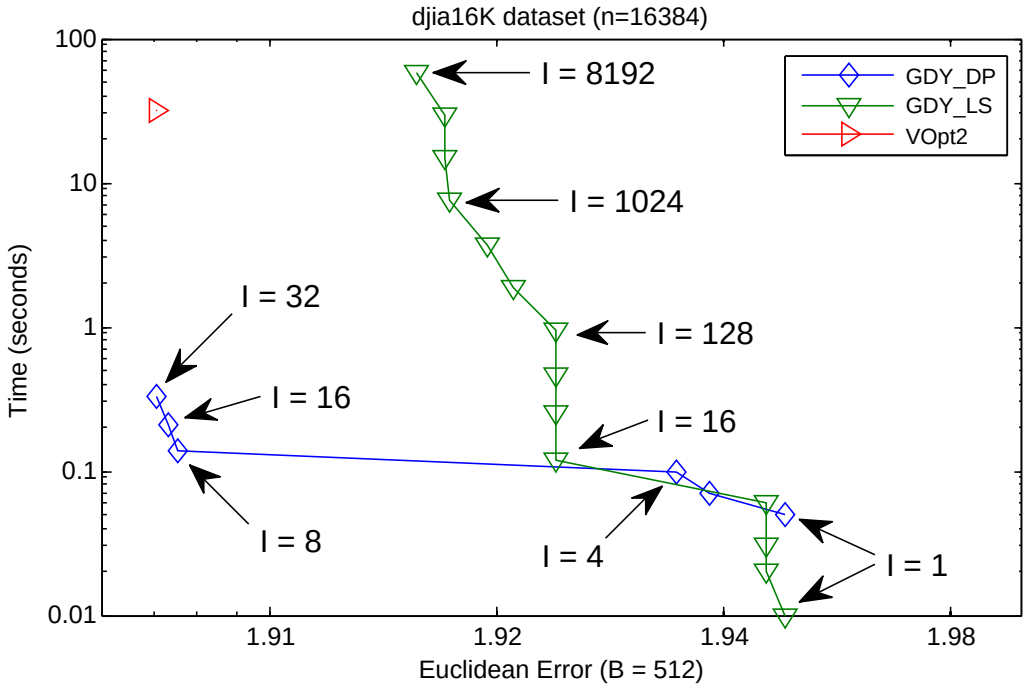


Figure 2.36: GDY_LS vs. GDY_DP, $B = 512$: DJIA

Figure 2.36 shows the performance of GDY_LS compared to GDY_DP. The pure local search, GDY_LS, takes many more iterations (hence, computation

time) when compared with GDY_DP which reaches the same quality as the optimal algorithm, VOpt2, in 32 iterations. We can see also that changing GDY_LS to a cleverer local search (perhaps utilizing some problem specific domain optimizations) can lower the number of iterations required, giving smaller times or lower error or both. The optimal algorithm thus serves as a catalyst for building variants of local search.

2.8 Conclusion

In this chapter we offer a fresh approach to sequence segmentation or histogram construction that addresses a critical gap in scalability versus quality which is important in justifying the resources spent, especially in environment with limited amount of resources available. To the best of our knowledge, it is the *first* work to develop segmentation algorithms that are *both* fast and scalable (i.e., linear) in terms of time efficiency, *and* effective in terms of the quality they achieve, as measured in terms of Euclidean error. Moreover, we have conducted the *first*, to our knowledge, experimental comparison of proposed heuristics and approximation schemes for sequence segmentation. In the future, we may want to expand the comparisons to include wavelet based histogram / segmentation.

Our best-performing method is based on an application of stochastic local search that generates sample boundaries which effectively capture the characteristics of the underlying data sequence. These sample boundaries are then used as input points for the dynamic-programming segmentation algorithm that recombines an optimal subset among them. In order to scale to very large data sequence, we processes the candidates in *batches*, so that its time complexity is kept constrained while maintaining its quality. By combining all these techniques, we addressed a Big Data problem in sequence segmentation where existing solutions become impractical to run as the data sequence grows beyond a certain size or fail to deliver satisfactory quality. In the next chapter, we will address another type of Big Data problem where the existing solution fails due to it not being robust.

Chapter 3

Robust Cracking

Scientific data tends to be very large both in terms of the number of tuples and its attributes. For example, a table in the SkyServer dataset has 466 columns and 270 million rows [64]. Moreover, new datasets may arrive periodically and the queries imposed on scientific data are very dynamic and unpredictable (i.e., it does not necessarily follow a pre-determined pattern, and may depend on the previous query result, or the queries may be arbitrary/exploratory). These characteristics pose as an interesting challenge in creating efficient query processing system (detailed in the next section).

Having limited resources, to process such scientific data, it is paramount to have algorithms that can conserve and spend resources wisely. One recent advances in database research has a philosophy of always do *just enough*, namely *Database Cracking* [53]. To conserve resources, cracking strictly process only the necessary/relevant data for the query at hand. However, as we shall see, this philosophy fails under dynamic query workloads because it does not consider the underlying properties of the data. Since cracking uses the queries as advice to reorganize the physical store of the data, blindly following the queries may lead to an unfavorable physical store. Thus, under dynamic query workloads, cracking fails to conserve resources and spends resources significantly more than it should.

Learning from the previous chapter, we propose to relax the philosophy and invest some resources to focus on the current properties of the data and factor in stochasticity. Our new *stochastic cracking* algorithms manage to maintain the sub-linear runtime complexity per query and achieve an overall efficient, effective, and robust behavior under dynamic and unpredictable query workloads. Our stochastic cracking outperforms the original cracking by two orders of magnitude faster on a real dataset and query workload. The rest of the chapter elaborates the details of the robustness problem in original cracking and the solutions. Our work on stochastic cracking is published in [46].

3.1 Database Cracking Background

The goal of database cracking is to a build self-organizing database system that will continuously and automatically refine its physical data store to rapidly improve future queries' response time (i.e., *adapt* to query workloads). It is achieved by incrementally building indexes as a side effect of query processing, taking the queries as advice to reorganize its physical data store. The philosophy behind database cracking is to always do *just enough* [53]. That is, it tries to minimize the amount of resources used to answer just the current query at hand.

To understand the motivation behind database cracking, we first briefly recap the two traditional approaches to indexing and tuning: offline indexing and online indexing.

Offline indexing assumes the users have the workload knowledge which can be sampled and analyzed (offline) from the past query logs or other knowledge about queries. Offline indexing also requires sufficient idle time to analyze and prepare the physical design before answering the queries. While this may work very well for most small to medium sized databases, it is problematic for large databases under dynamic environments where workload knowledge is minimal and idle time is a scarce resources. For example, in scientific databases, new data arrives on a daily or even hourly basis, while query patterns follow an exploratory path as the scientists try to interpret the data and understand the patterns observed; there is no time and knowledge to analyze and prepare a different physical design every hour or even every day. Furthermore, with the dynamic nature of the (exploratory) query workloads, any offline decision may soon become invalid.

Online indexing strategies aim to tackle the problem posed by such dynamic workloads. A number of recent efforts attempt to provide viable online indexing solutions[19, 97, 18, 81]. Their main common idea is to apply the basic concepts of offline analysis online: the system monitors its workload and performance while processing queries, probes the need for different indexes and, once certain thresholds are passed, triggers the creation of such new indexes and possibly drops old ones. However, online analysis may severely overload individual query processing during index creation and several queries may run without index support. Approaches such as soft indexes [81] try to exploit the scan of relevant data (e.g., by a select operator) and send this data to a full-index creation routine at the same time. This way, data to be indexed is read only once. Still, the problem remains that creating *full* indexes significantly penalizes individual queries.

In summary, traditional indexing presents three fundamental weaknesses: (a) the workload may have changed by the time we finish tuning; (b) there may be no time to finish tuning properly; and (c) there is no indexing support during

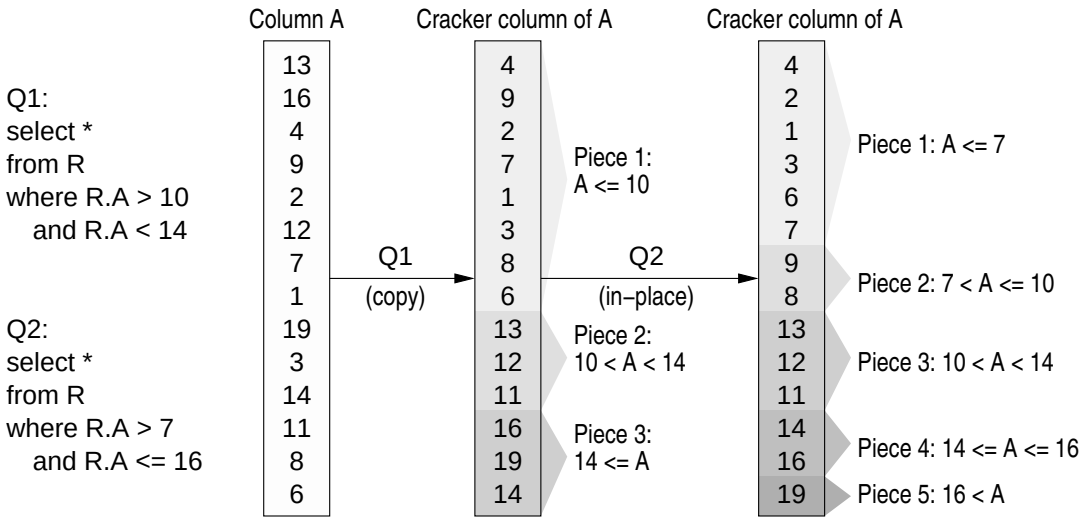


Figure 3.1: Cracking a column

tuning. These drawbacks motivate the invention of a novel *adaptive indexing* strategy, the prime example of which is *database cracking* [53].

Database cracking assumes no idle time is available for preparation and assumes that the query workload is unpredictable as in dynamic environments. The user queries are used as advice to continuously, incrementally (and partially) build the indexes as part of query processing and thus it requires minimal tooling or administrator effort to function. The cracking philosophy ensures that only those tables, columns, and key ranges that are queried are being optimized. The more often a key range is queried, the more its representation is optimized. Non-queried columns remain non-indexed, and non-queried key ranges are not optimized.

Figure 3.1 shows an example of database cracking in action. For simplicity, assume we have a column A which consists of integer values. The first query, Q1, arrives and would like to select a range of values between 10 and 14 (exclusive). The first time column A gets accessed, the contents of column A are copied once to a *cracker column* A. From here on, the cracker column A is used to answer the queries. The physical store of the cracker column A will be refined as the side effect of query processing. Initially, the cracker column A has no index. Thus to process Q1, the entire column must be examined to find the tuples that qualifies the query. In addition to that, the tuples will be reorganized on the cracker column A that uses Q1 as advice on how data should be stored. That is it will move all tuples that are less than or equal to 10 to the upper part (beginning) of the cracker column A and all tuples that are bigger or equal to 14 to the bottom part (end) of the cracker column A. This reorganization separates the column into three pieces. These pieces are stored as cracker indexes to speed-up the subsequent queries. The qualified tuples for Q1 are *clustered* in a contiguous area

in the middle piece (Piece 2). When the second query Q2 arrives, it can take advantage of the existing cracker indexes. Q2 only needs to examine and refine the first piece and last piece (i.e., it doesn't need to examine the entire column. In this case, it doesn't need to examine the second piece since we know that the second piece already qualifies) and enhances the cracker indexes further.

A cracking DBMS maintains cracker indexes showing which piece holds which value range, in a tree structure; original cracking uses AVL-trees [54]. These trees are meant to be bounded by a small depth by restricting the number of entries (or the minimum size of a cracking piece); thus, the cost of reorganizing data becomes the dominant part of the whole cracking cost. We can concretely identify this cost as the amount of data the system has to touch for every query, i.e., the number of tuples cracking has to analyze during a select operator. For example, in Figure 3.1 $Q1$ needs to analyze all tuples in the column in order to achieve the initial clustering, as there is no prior knowledge about the structure of the data. The second query, $Q2$, can exploit the knowledge gained by $Q1$ and avoid touching part of the data. With $Q1$ having already clustered the data into three pieces, $Q2$ needs to touch only two of those, namely the first and third piece. That is because the second piece created by $Q1$ already qualifies for $Q2$ as well. Generalizing the above analysis, we infer that, with such range queries (select operators), cracking needs to analyze at most two (*end*) pieces per query, i.e., the ones intersecting with the query's value range boundaries. As more pieces are created by every query that does not find an exact match, pieces become smaller.

The terminology "cracking" reflects the fact that the database is partitioned (cracked) into smaller and manageable pieces. Cracking gradually improves data access; as the size of the pieces that need to be examined gradually becomes smaller, it eventually leads to a significant speed-up in query processing, thus, *adapts* to the workload [54, 56]. Database cracking relies on a number of modern column-store design characteristics. Column-stores store data one column at a time in fixed-width dense arrays [83, 102, 17]. This representation is the same both on disk and in memory and allows for efficient physical reorganization of arrays. Similarly, column-stores rely on bulk and vector-wise processing. Thus, a select operator typically processes a single column in vector format at once, instead of whole tuples one at a time. In effect, cracking performs all physical reorganization actions efficiently in one go over a column. For example, the cracking select operator physically reorganizes the proper pieces of a column to bring all qualifying tuples (or values) in a contiguous area and then returns a view of this area as the result. To date, all work on cracking and adaptive indexing has focused on main memory environments; persistent data may be on disk but the working data set for a given query (operator in a column-store) should fit in

memory for efficient query processing. Cracking was proposed in the context of modern column-stores and has been hitherto applied for boosting the performance of the select operator [54], maintenance under updates [55], and arbitrary multi-attribute queries [56]. In addition, more recently these ideas have been extended to exploit a partition/merge -like logic [57, 40, 41].

3.1.1 Ideal Cracking Cost

The ideal performance comes when analyzing fewer tuples. Such a disposition is workload-dependent; it depends not only on the nature of queries posed but also on the *order* in which they are posed. As in the analysis of the quicksort algorithm, Crack achieves the best-case performance (assuming a full column is relevant for the total workload) if each query cracks a piece of the column in exactly two half pieces: the first query splits the column in two equally sized pieces; the second and third query split it in four equal pieces, and so on, resulting in a uniform clustering of the data and gradual improvement of access patterns.

Assuming the general case where the whole value range of a given column will be touched, then the following simple analysis shows the optimal performance and the cost of an arbitrary cracking query.

Say that each query performs a single two-piece cracking action. Then, each crack splits a piece of the column into two new pieces of the same size, while each subsequent query symmetrically cracks the column into smaller pieces. For example, the first query will crack the whole column into two equal pieces p_{1a} and p_{1b}, the second query will crack p_{1a} again into two equal pieces p_{2a} and p_{2b}, the third query will crack p_{1b} into two equal pieces p_{2c} and p_{2d}. The fourth query will crack p_{2a}, the fifth query will crack p_{2b} and so on. Then, the cost for a sequence of queries cracking a column of N values becomes as follows.

$$N + \frac{N}{2} + \frac{N}{2} + \frac{N}{4} + \frac{N}{4} + \frac{N}{4} + \frac{N}{4} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \frac{N}{8} + \ldots$$

The above cost is expressed in terms of data accesses, i.e., how many of the column values the crack algorithm needs to touch/analyze for each query. For example, the very first query needs to analyze every single value in the column and since cracking is a single pass algorithm the cost becomes N data accesses. Given that we assume that we always crack a piece in half, the second query will need to touch $\frac{N}{2}$ values and so on. This way, the cost of the i-th query in such a sequence becomes as follows.

$C_i = \frac{N}{2^{floor(log_2(i))}}$, where $log_2(i) = \frac{log(i)}{log(2)}$.

Similar to the randomized quicksort analysis, we argue that cracking is expected to perform reasonably close to ideal in random workload. Figure 3.2 shows a performance example where cracking (Crack) is compared against a full indexing approach (Sort), in which we completely sort the column with the first query. The data consists of $N = 10^8$ tuples of random unique integers in range from 0 to $N - 1$, while the query workload is completely random (the ranges requested have a fixed selectivity of 10 tuples per query but the actual bounds requested are random). This scenario assumes a dynamic environment where there is no workload knowledge or idle time in order to pre-sort the data, i.e., our very motivating example for adaptive indexing.

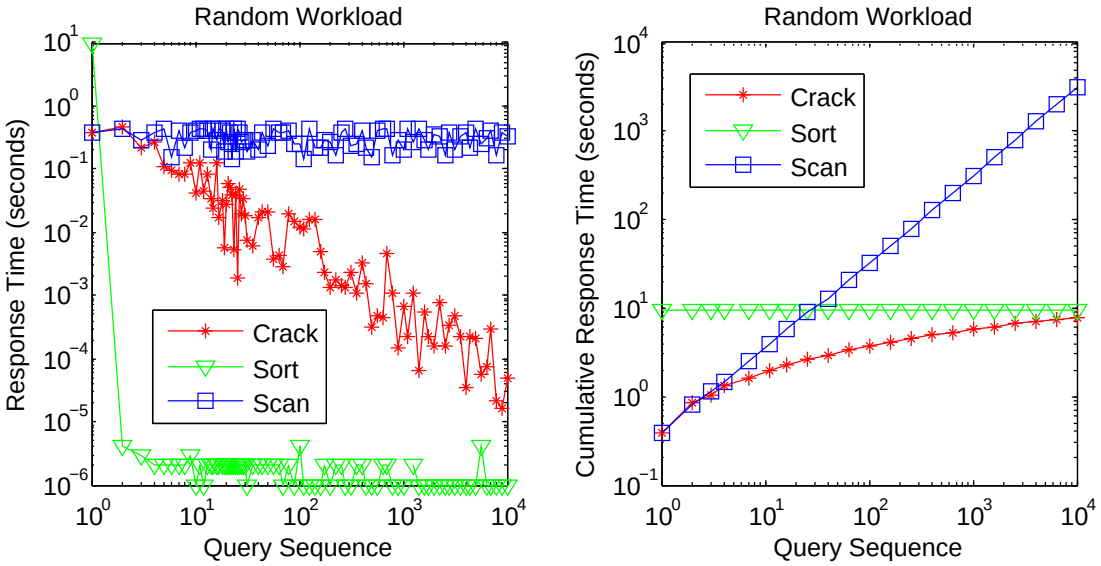


Figure 3.2: Basic Crack performance under Random Workload

As Figure 3.2 (left) shows, once the data is sorted with the first query, from then on the Sort performance is extremely fast as we only need to perform a binary search over the sorted column to satisfy each select operator request. Nevertheless, the problem is that we overload the first query. On the other hand, Crack continuously improves performance without penalizing individual queries. Eventually, its performance reaches the levels of Sort. We also compare against a plain Scan approach where data is always completely scanned. Naturally, this has a stable behavior; interestingly, Crack does not significantly penalize any query more than the default Scan approach. We emphasize that while Crack and Sort can simply return a view of the (contiguous) qualifying tuples, Scan has to materialize a new array with the result. Figure 3.2(right) shows in cumulative response time that while Crack has finished answering 10^4 queries, Sort has not yet answered a single query. Moreover, even after answering 10^4 queries, Sort has not amortized its initialization overhead over Crack. This result shows the

principal advantage of database cracking: its lightweight adaptation.

3.2 The Workload Robustness Problem

Existing cracking scheme interprets queries as a hint on how to organize the data store and the remainder of the data remains non-indexed until a query expresses interest therein. This cracking philosophy brings *instant and lightweight adaptation* to user query workloads. This section will show that the same philosophy could destroy the adaptive feature it set out to achieve.

Existing cracking schemes faithfully and obediently follow the hints provided by the queries in a workload, without examining whether these hints make good sense from a broader view. This approach fares quite well with random workloads, or workloads that expose consistent interest in certain regions of the data. However, in other realistic workloads, this approach can fail. For example, consider a workload where successive queries ask for consecutive items, as if they sequentially scan the value domain; we call this workload pattern *sequential*. Figure 3.4 shows an example of such a sequential workload among many others (see Figure 3.4 for the details). If we assume that the column has N tuples with unique integers, then the first query will cost N comparisons, the second query will cost $N - 20$, the third $N - 40$ and so on, causing such a workload to exhibit a very slow adaptation rate. By contrast, in the ideal case where the first query splits the column into two equal parts, the second query already had a reduced cost down to $N/2$.

Applying existing cracking methods on this sequential workload would result into repeatedly reorganizing large chunks of data with every query; yet this expensive operation confers only a minor benefit to subsequent queries. Thus, existing cracking schemes fail in terms of *workload robustness*. Such a workload robustness problem emerges with any workload that focuses in a specific area of the value domain at a time, leaving (large) unindexed data pieces that can cause performance degradation if queries touch this area later on. Such workloads occur in exploratory settings; for example, in scientific data analysis in the astronomy domain, scientists typically "scan" one part of the sky at a time through the images downloaded from telescopes.

Figure 3.3 shows the results with such a workload. As in Figure 3.2, we test Crack against Scan and Sort. The setup is exactly the same as before, i.e., the data in the column, the initial status, and the query selectivity are the same as in the experiment for Figure 3.2; the only difference is that this time queries follow the sequential workload. We observe that Sort and Scan are not affected by the kind of workload tested; their behavior with random and sequential workloads do

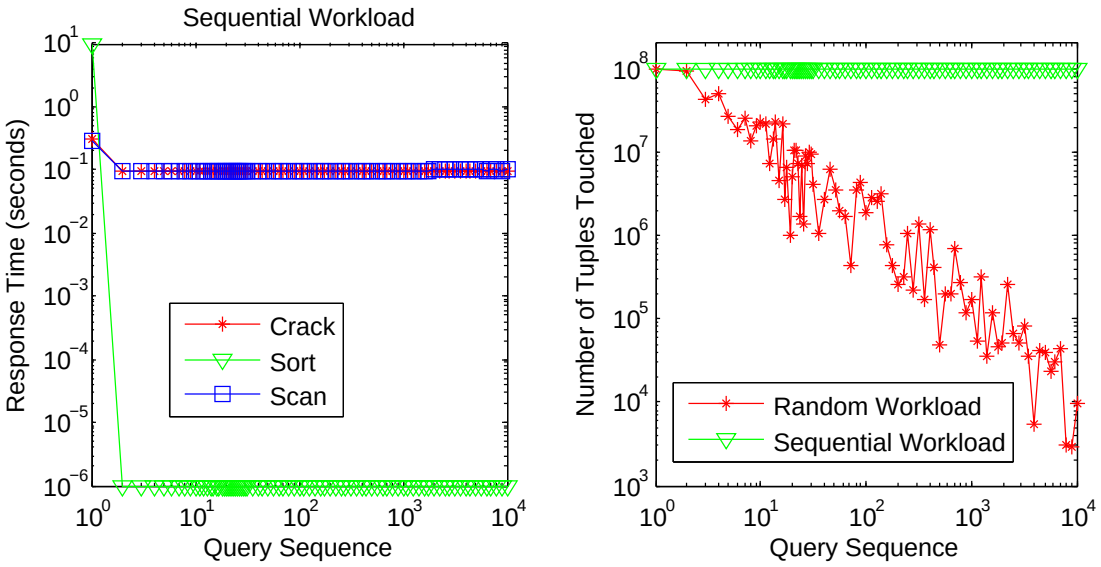
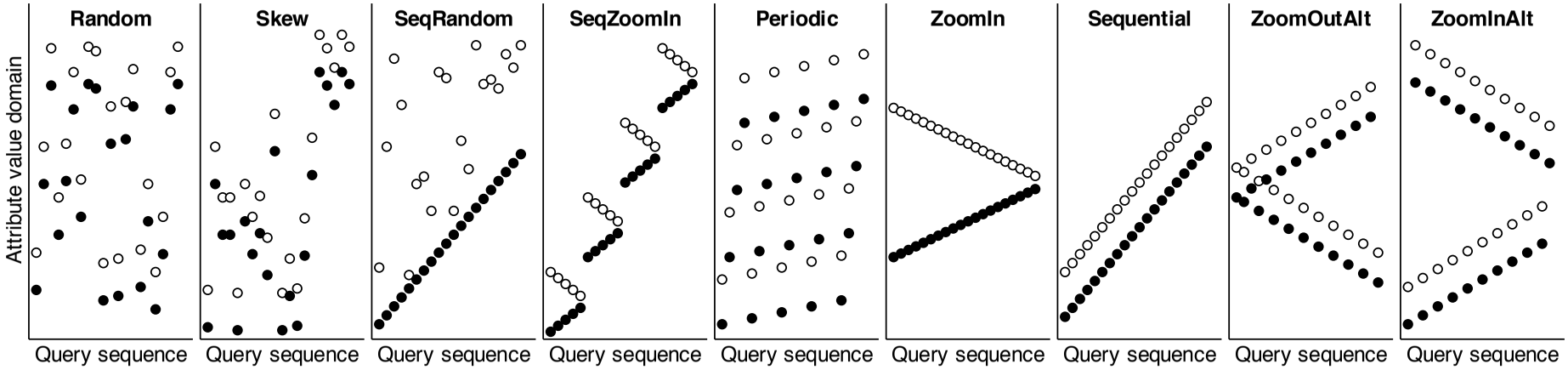


Figure 3.3: Crack loses its adaptivity in a Non-Random Workload

not deviate significantly from each other. This is not surprising, as the Scan will always scan N tuples no matter the workload, while the full indexing approach will always pay for the complete sort with the first query and then exploit binary search. A slight improvement observed in the Scan performance is due to the short-circuiting in the if statement checking for the requested range. Likewise, there is slight improvement for the Sort strategy after the first query due to caching effects of the binary search in successive short ranges. By contrast, Figure 3.3 clearly shows that Crack fails to deliver the performance improvements seen for the random workload in Figure 3.2. Now its performance does not outperform that of Scan, whereas with the random workload performance improved significantly already after a handful of queries.

To elaborate on this result, Figure 3.3(right) shows the number of tuples each cracking query needs to touch with these two workloads. With the sequential workload, Crack touches a large number of tuples, which falls only negligibly as new queries arrive, whereas with the random workload the number of touched tuples drops swiftly after only a few queries. With less data to analyze, performance improves rapidly.

In the rest of the chapter, we lay out our cracking schemes that satisfy the workload-robustness imperative. To do so, we re-examine the underlying assumptions of existing schemes and propose a significantly more resilient alternative. We show that original cracking relies on the *randomness* of the workloads to converge well; we argue that, to succeed with non-random workloads, cracking needs to introduce randomness on its own. Our proposal introduces arbitrary and random, or *stochastic*, elements in the cracking process; each query is still taken as a hint on how to reorganize the data, albeit in a *lax* manner that allows for



Workload	**[low bound, high bound) for i-th query sequence**
Random:	[a, a+S), where a = R%(N-S)
Skew:	[a, a+S), where a = R%(N*0.8-S) for i < Q*0.8, otherwise a = N*0.8 + R%(N*0.2-S)
SeqRandom:	[i*J, i*J+R%(N-i*J))
SeqZoomIn:	[L+K, L+W-K), where L = (i div 1000)*W, K=(i%1000)*J
Periodic:	[a, a+S), where a = (i*J)%(N-S)
ZoomIn:	[N/2-W/2+i*J, N/2+W/2-i*J)
Sequential:	[a, a+S), where a = i*J
ZoomOutAlt:	[a, a+S), where a = x*i*J + M, M = N/2, x = (-1)^i
ZoomInAlt:	[a, a+S), where a = x*i*J + (N-S)*(1-x)/2, x = (-1)^i

Variables:

J = jump factor
S = query selectivity
Q = number of query sequences
R = generates a random integer
W = initial width

Notes: The dataset is N=10^8 unique integers in range [0,N).
Operator % is for modulo, **div** is for integer division.
The workloads in the figure are ordered from left to right by Stochastic Crack's gain over Crack's in increasing order.
SeqReverse, ZoomOut, SeqZoomOut workloads are identical to Sequential, ZoomIn, SeqZoomIn run in reverse query sequence.
SkewZoomOutAlt is ZoomOutAlt with M = N*9/10.

Figure 3.4: Various workloads patterns
A query is represented by a pair of black and white circle. The black is the low bound of the query and the white circle is the high bound of the query.

reorganization steps not explicitly dictated by the query itself. While we introduce such auxiliary actions, we also need to maintain the lightweight character of existing cracking schemes. To contain the overhead brought about by stochastic operations, we introduce *progressive* cracking, in which a single cracking action is completed *collaboratively* by multiple queries instead of a single one. Our experimental study shows that stochastic cracking preserves the benefits of original cracking schemes, while also expanding these benefits to a large variety of realistic workloads on which original cracking fails.

Overall, the workload robustness requirement is a major challenge for future database systems [38]. While we know how to build well-performing specialized systems, designing systems that perform well over a broad range of scenarios and environments is significantly harder. We emphasize that this workload robustness imperative does not imply that a system should perform all conceivable tasks efficiently; it is accepted nowadays that "one size does not fit all" [101]. However, it does imply that a system's performance should not deteriorate after changing a minor detail in its input or environment specifications. The system should maintain its performance and properties when faced with such changes. The whole spectrum of database design and architecture should be re-investigated with workload robustness in mind [38], including, e.g., optimizer policies and low-level operator design.

3.3 Stochastic Cracking

Having discussed the robustness problem, we now present our proposal in a series of incrementally more sophisticated algorithms that aim to achieve the desired workload robustness while maintaining the adaptability of existing cracking schemes.

In Section 3.2, we have shown that the cost of a query (select operator) with cracking depends on the amount of data that needs to be analyzed for physical reorganization. The sequential workload which we have used as an example to demonstrate the weakness of original cracking, forces cracking to repeatedly analyze large data portions for consecutive queries.

This effect is due to the fact that cracking treats each query as a hint on how to reorganize data in a *blinkered* manner: it takes each query as a literal instruction on what data to index, without looking at the bigger picture. It is thanks to this literalness that cracking can instantly adapt to a random workload; yet, as we have shown, this literal character can also be a liability.

With a non-ideal workload, strictly adhering to the queries and reorganizing the array so as to collect the query result, amounts to an inefficient quicksort-like

operation; small successive portions of the array are clustered, one after the other, while leaving the rest of the array unaffected. Each new query, having a bound inside the unindexed area of the array, reanalyzes this area all over again.

To address this problem, we venture to drop the strict requirement in original cracking that each individual query be literally interpreted as a re-organization suggestion. Instead, we want to force reorganization actions that are not strictly driven by what a query requests, but are still beneficial for the workload at large.

To achieve this outcome, we propose that reorganization actions be *partially* driven by what queries want, and *partially* arbitrary in character. We name the resulting cracking variant *stochastic*, in order to indicate the arbitrary nature of some of its reorganization actions. We emphasize that our new variant should not totally forgo the *query-driven* character of original cracking. An extreme stochastic cracking implementation could adopt a totally arbitrary approach, making random reorganizations along with each query (we discuss such naive cracking variants in Section 3.4). However, such an approach would discard a feature of cracking that is worth keeping, namely the capacity to adapt to a workload without significant delays. Besides, as we have seen in Figure 3.2, cracking barely imposes any overhead over the default scan approach; while the system adapts, users do not notice significantly slower response times; they just observe faster reaction times later. Our solution should maintain this lightweight property of original cracking too.

Our solution is a sophisticated intermediary between totally query-driven and totally arbitrary reorganization steps performed with each query. It maintains the lightweight and adaptive character of existing cracking, while extending its applicability to practically any workload. In the rest of this section, we present techniques that try to strike a balance between (a) adding auxiliary reorganization steps with each query, and (b) remaining lightweight enough so as not to significantly (if at all) penalize individual queries.

All our stochastic cracking algorithms are proposed as replacements for the original cracking physical reorganization algorithm [54]. The various cracking algorithms proposal are summarized in Table 3.1. The stochastic cracking algorithms are underlined while the rest are used for comparisons. From a high level point of view, nothing changes, i.e., stochastic cracking maintains the design principles for cracking a column-store. As in original cracking [54], in stochastic cracking the select operator physically reorganizes an array that represents a single attribute in a column-store so as to introduce range partitioning information. Meanwhile, a tree structure maintains structural knowledge, i.e., keeps track of which piece of the clustered array contains which value range. This way, the general setting is as follows. As new queries arrive, the select operators therein

trigger cracking actions. Each select operator requests for a range of values on a given attribute (array) and the system reacts by physically reorganizing this array, if necessary, and collecting all qualifying tuples in a continuous area. The difference we introduce with stochastic cracking is that, instead of passively relying on the workload to stipulate the *kind* and *timing* of reorganizations taking place, it exercises more control over these decisions.

Acronym	*Description*	*Section*
DDC	Data Driven Center ($O(log(N))$ auxiliary cracks)	3.3.1
DDR	Data Driven Random ($O(log(N))$ auxiliary cracks)	3.3.2
DD1C	DDC with at most 1 auxiliary crack	3.3.3
DD1R	DDR with at most 1 auxiliary crack	3.3.3
MDD1R	Materialized DD1R (pure stochastic cracking)	3.3.4
P(X)%	Progressive MDD1R limited to X% reorganizations	3.3.5
Mon(X)	Triggers stochastic crack on X touches to a piece	3.3.6
Sel(X)%	Perform stochastic cracking X% of the time	3.3.6
Naive (X)th	Perform a naive random crack on every X-th query	3.4.1
Naive (X)R	Perform X naive random cracks on every query	3.4.1

Table 3.1: Cracking Algorithms

3.3.1 Data Driven Center (DDC)

Our first algorithm, the *Data Driven Center* algorithm (DDC), exercises its own decision-making without using random elements; we use it as a baseline for the subsequent development of its genuinely stochastic variants. The motivation for DDC comes from our analysis of the ideal cracking behavior in Section 3.1.1; ideally, each reorganization action should split the respective array piece in *half*, in a quicksort-like fashion. DDC *recursively* halves relevant pieces on its way to the requested range, introducing several new pieces with each new query, especially for the first queries that touch a given column. The term "*Center*" in its name denotes that it always tries to cut pieces in half.

The other component in its name, namely "*Data Driven*", contrasts it to the *query-driven* character of default cracking; if a query requests the range $[a, b]$, default cracking reorganizes the array based on $[a, b]$ regardless of the actual data. By contrast, DDC takes the data into account. Regardless of what kind of query arrives, DDC always performs specific data-driven actions, *in addition* to query-driven actions. The query-driven mentality is maintained, as otherwise the algorithm would not provide good adaptation.

Given a query in $[a, b]$, DDC *recursively* halves the array piece where $[a, b]$ falls, until it reaches a point where the size of the resulting piece is sufficiently small.

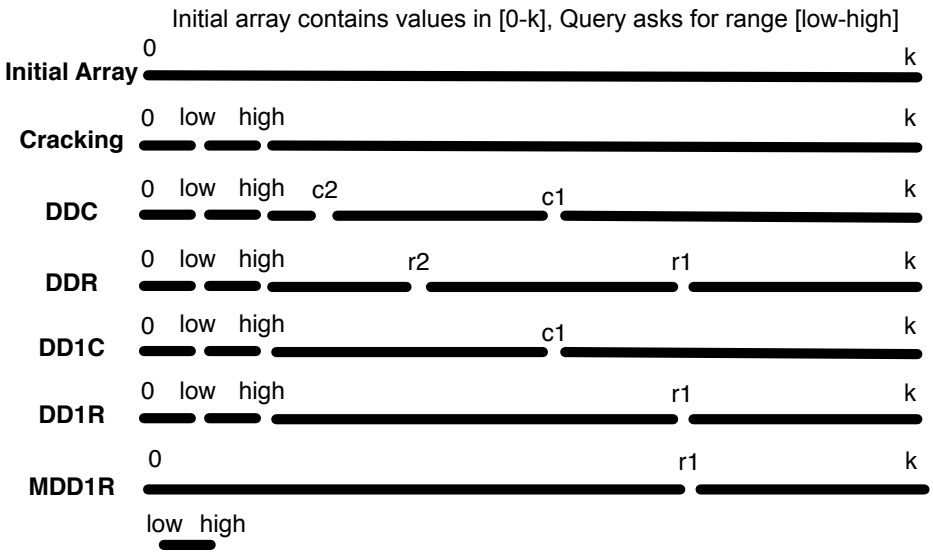


Figure 3.5: Cracking algorithms in action

Then, it cracks this piece based on $[a, b]$. As with original cracking, a request for $[a, b]$ in an already cracked column will in general result in two requests/cracks; one for $[a,)$ and one for $(, b]$ (as for Q2 in Fig. 3.1).

A high-level example for DDC is given in Figure 3.5. This figure shows the end result of a simplifying example of data reorganization with the various stochastic cracking algorithms that we introduce, as well as with original cracking. An array, initially uncracked, is queried for a value range in [low, high]. The initially uncracked array, as well as the separate pieces created by the various cracking algorithms, is represented by continuous lines. We emphasize that these are only *logical* pieces, since all values are still stored in a single array; however, cracking identifies (and incrementally indexes) these pieces and value ranges.

As Figure 3.5 shows, original cracking reorganizes the array solely based on [low, high], i.e., exactly what the query requested. On the other hand, DDC introduces more knowledge; it first cracks the array on c1, then on c2, and only then on [low, high]. The bound c1 represents the median that cuts the complete array into two pieces with equal number of tuples; likewise, c2 is the median that cuts the left piece into two equal pieces. Thereafter, the newly created piece is found to be small enough; DDC stops searching for medians and *cracks* the piece based on the query's request. For the sake of simplicity, in this example both low and high fall in the same piece and only two iterations are needed to reach a small enough piece size. In general, DDC keeps cutting in half pieces until the minimum allowed size is reached. In addition, the request for [low, high] is evaluated as two requests, one for each bound, as in general each of the two bounds may fall in a different piece.

Figure 3.6 gives the DDC algorithm. Each query, DDC(C,a,b), attempts to

Algorithm DDC(C, a, b)
Crack array C on bounds a, b.
1. $positionLow$ = ddc_crack(C, a)
2. $positionHigh$ = ddc_crack(C, b)
3. $result$ = createView(C,$positionLow, positionHigh$)

function ddc_crack(C, v)
4. Find the piece $Piece$ that contains value v
5. $pLow = Piece$.firstPosition()
6. $pHgh = Piece$.lastPosition()
7. **while** ($pHgh$ - $pLow >$ CRACK_SIZE)
8. $pMiddle = (pLow+pHgh)$ / 2;
9. Introduce crack at $pMiddle$
10. **if** ($v < C[pMiddle]$) $pHgh = pMiddle$
11. **else** $pLow = pMiddle$
12. $position$=crack($C[pLow, pHgh]$,v)
13. $result$=$position$

Figure 3.6: The DDC algorithm

introduce at least two cracks: on a and on b on column C. At each iteration, it may introduce (at most $log(N)$) further cracks. Function ddc_crack describes the way DDC cracks for a value v. First, it finds the piece that contains the target value v (Lines 4-6). Then, it recursively splits this piece in half while the range of the remaining relevant piece is bigger than CRACK_SIZE (Lines 7-11). Using order statistics, it finds the median M and partitions the array according to M in linear time (Line 9).

For ease of presentation, we avoid the details of the median-finding step in the pseudocode; the general intuition is that we keep reorganizing the piece until we hit the median, i.e., until we create two equal-sized pieces. At first, we simply cut the value *range* in half and try to crack based on the presumed median. Thereafter, we continuously adjust the bounds until we hit the correct median. The median-finding problem is a well-studied problem in computer science, with approaches such as BFPRT [16] providing linear complexity. We use the Introselect algorithm [88], which provides a good worst-case performance by combining quickselect with BFPRT. After the starting piece has been split in half, we choose the half-piece where v falls (Lines 10-11). If that new piece is still large, we keep halving, otherwise we proceed with regular cracking on v and return the final index position of v (Lines 12-13).

In a nutshell, DDC introduces several data-driven cracks until the target piece is small enough. The rationale is that, by halving pieces, we *contain* the cases unfavorable to cracking (i.e., the repeated scans) to small pieces. Thus, the

repercussions of such unfavorable cases become negligible. We found that the size of L1 cache as piece size threshold provides the best overall performance.

Still, DDC is also query-driven, as it introduces those cracks only on its path to find the requested values. As seen in Lines 7-11 of Figure 3.6, it recursively cracks those pieces that contain the requested bound, leaving the rest of the array unoptimized until some other query probes therein. This logic follows the original cracking philosophy, while inseminating it with data-driven elements for the sake of workload robustness. We emphasize that DDC preserves the original cracking interface and column-store requirements; it performs the same task, but adds extra operations therein. As Figure 3.5 shows, DDC collects all qualifying tuples in a piece of [low, high], as original cracking does.

3.3.2 Data Driven Random (DDR)

The DDC algorithm introduced several of the core features and philosophy of stochastic cracking, without employing randomness. The type of auxiliary operations employed by DDC is center cracks, always pivoted on a piece's median for optimal partitioning. However, finding these medians is an expensive and data-dependent operation; it burdens individual queries with high and unpredictable costs. It is critical for cracking, and any adaptive indexing technique, to achieve a low initialization footprint. Queries should not be heavily, if at all, penalized while adapting to the workload. Heavily penalizing a few queries would defeat the purpose of adaptation [39].

Original cracking achieves this goal by performing partitioning and reorganization following only what queries ask for. Still, we have shown that this is not enough when it comes to workload robustness. The DDC algorithm does more than simply following the query's request and thus introduces extra costs. The rest of our algorithms try to strike a good tradeoff between the auxiliary knowledge introduced per query and the overhead we pay for it.

Our first step in this direction is made with the *Data Driven Random* algorithm (DDR), which introduces random elements in its operation. DDR differs from DDC in that it relaxes the requirement that a piece be split exactly in half. Instead, it uses *random* cracks, selecting random pivots until the target value v fits in a piece smaller than the threshold set for the maximum piece size. Thus, DDR can be thought of as a single-branch quicksort. Like quicksort, it splits a piece in two, but, unlike quicksort, it only recurses into one of the two resulting pieces. The choice of that piece is again query-driven, determined by where the requested values fall.

Figure 3.5 shows an example of how DDR splits an array using initially a

random pivot r1, then recursively splits the new left piece on a random pivot r2, and finally cracks based on the requested value range to create piece [low, high]. Admittedly, DDR creates less well-chosen partitions that DDC. Nevertheless, in practice, DDR makes substantially less effort to answer a query, since it does not need to find the correct medians as DDC does, while at the same time it does add auxiliary partitioning information in its randomized way. In a worst-case scenario, DDR may get very unlucky and degenerate to $O(N^2)$ cost; still, it is expected that in practice the randomly chosen pivots will quickly lead to favorable piece sizes.

3.3.3 Restricted Data Driven (DD1C and DD1R)

By recursively applying more and more reorganization, both DDC and DDR manage to introduce indexing information that is useful for subsequent queries. Nevertheless, this recursive reorganization may cause the first few queries in a workload to suffer a considerably high overhead in order to perform these auxiliary operations. As we discussed, an adaptive indexing solution should keep the cost of initial queries low [39]. Therefore, we devise two variants of DDC and DDR, which eschew the recursive physical reorganization. These variants perform *at most one* auxiliary physical reorganization. In particular, we devise algorithm DD1C, which works as DDC, with the difference that, after cutting a piece in half, it simply cracks the remaining piece where the requested value is located regardless of its size. Likewise, algorithm DD1R works as DDR, but performs only one random reorganization before it resorts to plain cracking.

DD1C corresponds to the pseudocode description in Figure 3.6, with the modification that the while statement in Line 7 is replaced by an if statement. Figure 3.5 shows a high-level example of DD1C and DD1R in action. The figure shows that DD1C cuts only the first piece based on bound c1 and then cracks on [low, high]; likewise, DD1R uses only one random pivot r1. In both cases, the extra steps of their fully recursive siblings are avoided.

3.3.4 Materialized Data Driven Random (MDD1R)

Algorithms DD1C and DD1R try to reduce the initialization overhead of their recursive siblings by performing only one auxiliary reorganization operation, instead of multiple recursive ones. Nevertheless, even this one auxiliary action can be visible in terms of individual query cost, especially for the first query or the first few queries in a workload sequence. That is so because the first query will need to crack the whole column, which for a new workload trend will typically be completely uncracked.

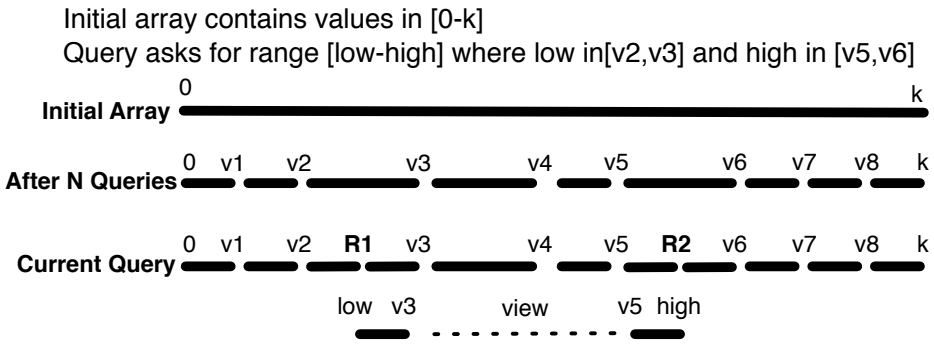


Figure 3.7: An example of MDD1R

Motivated to further reduce the initialization cost, we devise algorithm MDD1R, where "M" stands for *materialization*. This algorithm works like DD1R, with the difference being that it does *not* perform the final cracking step based on the query bounds. Instead, it materializes the result in a new array. Thus, all the cracks performed are pure stochastic cracks (no cracks based on the query bounds).

DD1R and DD1C perform two cracking actions: (1) one for the center or random pivot cracking and (2) one for the query bounds. In contrast, regular cracking performs a single cracking action, only based on the query bounds. Our motivation for MDD1R is to reduce the stochastic cracking costs by eschewing the final cracking operation. Prudently, we do not do away with the random cracking action, as this is the one that we have introduced aiming to achieve workload robustness. Thus, we drop the cracking action that follows the query bounds. However, we still have to answer the current query (select operator). Therefore, we choose to materialize the result in a new array, just like a plain (non-cracking) select operator does in a column-store. To perform this materialization step efficiently, we integrate it with the random cracking step: we detect and materialize qualifying tuples *while* cracking a data piece based on a random pivot. Otherwise, we would have to do a second scan after the random crack, incurring significant extra cost. Besides, we materialize only when necessary, i.e., we avoid materialization altogether when a query exactly matches a piece, or when qualifying tuples do not exist at the end pieces.

Figure 3.5 shows high-level view of MDD1R in action. Notably, MDD1R performs the same random crack as DD1R, but does not perform the query-based cracking operation as DD1R does; instead, it just materializes the result tuples. A pseudocode for the MDD1R algorithm is shown in Figure 3.8.

Figure 3.7 illustrates a more detailed example on a column that has already been cracked by a number of preceding queries. In general, the two bounds that define a range request in a select operator fall in two different pieces of an already cracked column. MDD1R handles these two pieces independently; it first operates

Algorithm MDD1R(C, a, b)
Crack array C on bounds a, b.
1. Find the piece $P1$ that contains value a
2. Find the piece $P2$ that contains value b
3. **if** ($P1 == P2$)
4. $result$ = split_and_materialize($P1$,a,b)
5. **else**
6. $res1$ = split_and_materialize($P1$,a,b)
7. $res2$ = split_and_materialize($P2$,a,b)
8. $view$ = createView($C, P1.lastPos$+1, $P2.firstPos$-1)
9. $result$ = concat($res1, view, res2$)

function split_and_materialize(Piece,a,b)
10. L=Piece.firstPosition
11. R=Piece.lastPosition
12. $result$=newArray()
13. $X = C[L$ + rand()%(R-L+1)]
14. **while** ($L <= R$)
15. **while** ($L <= R$ and $C[L] < X$)
16. **if** ($a <= C[L]$ && $C[L] < b$) $result.Add(C[L])$
17. $L = L + 1$
18. **while** ($L <= R$ and $C[R] >= X$)
19. **if** ($a <= C[R]$ && $C[R] < b$) $result.Add(C[R])$
20. $R = R$ - 1
21. **if** ($L < R$) **swap**($C[L]$,$C[R]$)
22. Add crack on X at position L

Figure 3.8: The MDD1R algorithm

solely on the leftmost piece intersecting with the query range, and then on the rightmost piece, introducing one random crack per piece. In addition, notice that the extra materialization is only *partial*, i.e., the middle qualifying pieces which are not cracked are returned as a view, while only any qualifying tuples from the end pieces need to be materialized. This example also highlights the fact that MDD1R does *not* forgo its query-driven character, even while it eschews query-based cracking per se; it still uses the query bounds to decide *where* to perform its random cracking actions. In other words, the choice of the pivots is random, but the choice of the pieces of the array to be cracked is query-driven.

We do a number of optimizations over the algorithm shown in Figure 3.8. For example, we reduce the number of comparisons by having specialized versions of the split_and_materialize method. For instance, a request on $[a, b)$ where a and b fall in different pieces, $P1$ and $P2$, will result in two calls, one in $P1$ only, checking for $v > a$, and one on $P2$ only, checking for $v \leq b$.

3.3.5 Progressive Stochastic Cracking (PMDD1R)

Our next algorithm, *Progressive* MDD1R (*PMDD1R*) is an even more incremental variant of MDD1R which further reduces the initialization costs. The rationale behind cracking is to build indexes incrementally, as a sequence of several small steps. Each such step is triggered by a single query, and brings about physical reorganization of a column. With PMDD1R we introduce the notion of *progressive* cracking; we take the idea of incremental indexing one step further, and extend it even at the individual cracking steps themselves. PMDD1R completes each cracking operation incrementally, in several partial steps; a physical reorganization action is completed by a sequence of queries, instead of just a single one. The goal is to significantly reduce the cost of cracking and make it as invisible as possible for the end user.

In our design of progressive cracking, we introduce a restriction on the number of physical reorganization actions a single query can perform on a given piece of an array; in particular, we control the number of *swaps* performed to change the position of tuples.

The resulting algorithm is even more lightweight than MDD1R; like MDD1R, it also tries to introduce a single random crack per piece (at most two cracks per query) and materializes part of the result when necessary. The difference of PMDD1R is that it only *gradually* completes the random crack, as more and more queries touch (want to crack) the same piece of the column. For example, say a query q_1 needs to crack piece p_i. It will then start introducing a random crack on p_i, but will only complete part of this operation by allowing $x\%$ swaps to be completed ($x\%$ of the number of tuples in the piece); q_1 is fully answered by materializing all qualifying tuples in p_i. Then, if a subsequent query q_2 needs to crack p_i as well, the random crack initiated by q_1, *resumes* while executing q_2. Thus, PMDD1R is a generalization of MDD1R; MDD1R is PMDD1R with allowed swaps $x = 100\%$.

We emphasize that the restrictive parameter of the number of swaps allowed per query can be configured as a percentage of the number of tuples in the current piece to be cracked. We will study the effect of this parameter later. In addition, progressive cracking occurs only as long as the targeted data piece is bigger than the L2 cache, otherwise full MDD1R takes over. This provision is necessary in order to avoid slow convergence; we want to use progressive cracking only on large array pieces where the cost of cracking may be significant; otherwise, we prefer to perform cracking as usual so as to reap the benefits of fast convergence.

3.3.6 Selective Stochastic Cracking

Another alternative to reduce the overhead of stochastic actions is to *selectively eschew* stochastic cracking for some queries; such queries are answered using original cracking. One approach, which we call Det50%, deterministically applies stochastic cracking 50% of the time, i.e., only every other query. Still, as we will see, this approach encounters problems due to its deterministic elements, which forsake the robust probabilistic character of stochastic cracking. We propose the probabilistic variant, Sel50%, in which the choice of whether to apply stochastic cracking or original cracking for a given query is itself a probabilistic one.

In addition to switching between original and stochastic cracking in a periodic or random manner, we also design a monitoring approach, Mon$X$. Mon$X$ initiates query processing via original cracking but it also logs all accesses in pieces of a crack column. Each piece has a crack counter that increases every time this piece is cracked. When a new piece is created it inherits the counter from its parent piece. Once the counter for a piece p reaches a threshold X, then the next query that touch the piece p will use stochastic cracking to crack p, while resetting its counter. This way, Mon$X$ monitors all actions on individual pieces and applies stochastic cracking only when necessary and only on problematic data areas with frequent accesses.

Finally, an alternative selective stochastic cracking approach triggers stochastic cracking based on size parameters, i.e., switching from stochastic cracking to original cracking for all pieces in a column which become smaller than L1 cache; within the cache the cracking costs are minimized.

3.4 Experimental Analysis

In this section we demonstrate that Stochastic Cracking solves the workload robustness problem of original cracking.

We implemented all our algorithms in C++, using the C++ Standard Template Library for the cracker indices. All experiments ran on an Intel(R) Core(TM) i7 CPU 960 @ 3.20GHz machine with 12GB RAM running Fedora 12 (64-bit). As in past adaptive indexing work, our experiments are all main-memory resident, targeting modern main-memory column-store systems. We use several synthetic workloads as well as a real workload from the scientific domain. The synthetic workloads we use are presented in Figure 3.4. For each workload, the figure illustrates graphically and mathematically how sequences of queries touch the attribute value domain of a single column. The same setup is used as in Section 3.1.1; that is the initial values in the column consists of $N = 10^8$ random unique

integers in range 0 to $N - 1$.

3.4.1 Stochastic Cracking under Sequential Workload

We first study the behavior of Stochastic Cracking on a sequential workload. Figure 3.9 shows the results. Each graph depicts the cumulative response time, for one or more of the Stochastic Cracking variants, over the query sequence, in logarithmic axes. In addition, each graph shows the plot for original cracking and full indexing (Sort) so as to put the results in perspective. For plain cracking and Sort, the performance is identical to the one seen in Section 3.1.1: Sort has a high initial cost and then provides good search performance, while original cracking fails to improve.

Figure 3.9(a) depicts the results for DDR and DDC. Our first observation is that both Stochastic Cracking variants manage to avoid the bottleneck that original cracking falls into. They quickly improve their performance and converge to response times similar to those of Sort, producing a quite flat cumulative response time curve. This result demonstrates that, auxiliary reorganization actions can dispel the pathological effect of leaving large portions of the data array completely unindexed.

Comparing DDC and DDR to each other, we observe that DDR carries a significantly smaller footprint regarding its initialization costs, i.e., the cost of the first few queries that carry an adaptation overhead. In the case of DDC, this cost is significantly higher than that of plain cracking (we reiterate that the time axis is logarithmic). This drawback is due to the fact that DDC always tries to find medians and recursively cut pieces into halves. DDR avoids these costs as it uses random pivots instead. Thus, the cost of the first query with DDR is roughly twice faster than that of DDC, and much closer to that of plain cracking.

In order to demonstrate the effect of the piece size chosen as a threshold for Stochastic Cracking (i.e., if the size of the piece is bigger than the threshold X, it performs stochastic cracking otherwise the original cracking is performed). Figure 3.9(f) shows how it affects DDC. Piece size X = L1 cache size provides the best option to avoid cracking actions deemed unnecessary; larger threshold sizes cause performance to degrade due to the increased access costs on larger uncracked pieces. For a threshold even bigger than L2 cache size, performance degrades significantly as the access costs are substantial.

Figure 3.9(b) depicts the behavior of DD1R and DD1C. As with the case of DDR and DDC, DD1R similarly outperforms DD1C by avoiding the costly median search. Furthermore, by observing Figure 3.9(a) and (b), we see that the more lightweight Stochastic Cracking variants (DD1R and DD1C) reduce the

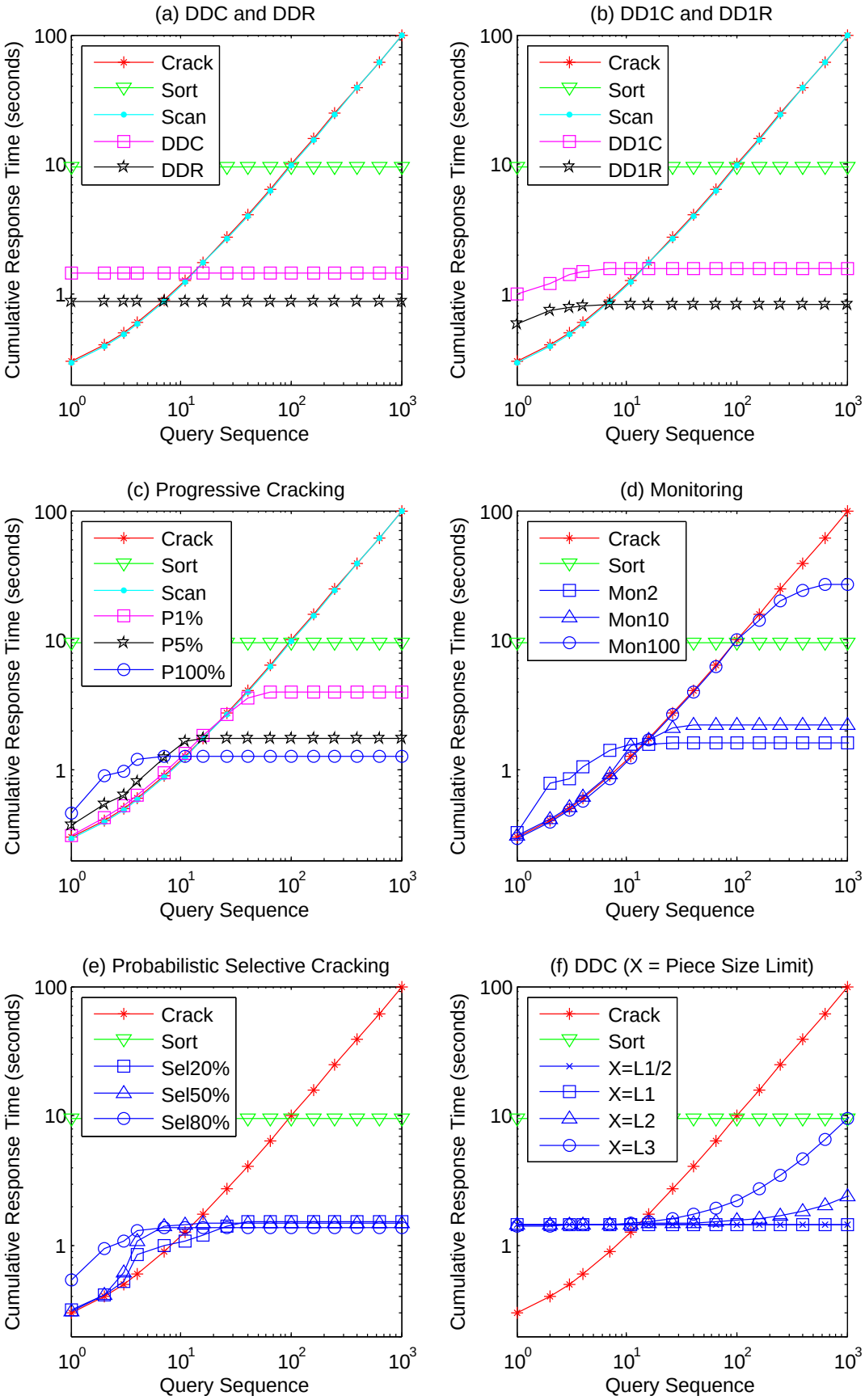


Figure 3.9: Stochastic Cracking under Sequential Workload

initialization overhead compared to their heavier counterparts (DDC and DDR). This is achieved by reducing the number of cracking actions performed with a single query. Naturally, this overhead reduction affects convergence, hence DDR and DDC (Figure 3.9(a)) converge very quickly to their best-case performance (i.e., their curves flatten) while DD1R and DD1C (Figure 3.9(b)) require a few more queries to do so (around 10). This extra number of queries depends on the data size; with more data, more queries are needed to index the array sufficiently well.

Figure 3.9(c) depicts the performance of progressive Stochastic Cracking, as a function of the amount of reorganization allowed. For instance, P5% allows for 5% of the tuples to be swapped per query. P100% is the same as MDD1R, imposing no restrictions. The more we constrain the amount of swaps per query, the more lightweight the algorithm becomes; thus, P1% achieves a first query performance similar to that of original cracking. Eventually (in this case, after 20 queries), the performance of P1% improves and then quickly converges (i.e., the curve flattens). The other progressive cracking variants obtain faster convergence as they impose fewer restrictions, hence their index reaches a good state much more quickly. Besides, especially in the case of the 5% variant, this relaxation of restrictions does not have a high impact on initialization costs. In effect, by imposing only a minimal initialization overhead, and without a need for workload knowledge or a priori idle time, progressive Stochastic Cracking can tackle this pathological workload.

Figure 3.9(d) shows the performance of Mon$X$ as described in Section 3.3.6. Mon2, Mon10, Mon100 will perform stochastic crack if the piece counter reaches 2, 10, and 100 respectively. The figure shows that the bigger the threshold, the worse the performance. The obvious reason is that before the piece counter reaches the threshold, the first few queries suffer in the same way with the original crack. This suggests that under pathological workload, it is prudent to always perform stochastic cracking.

Figure 3.9(e) shows the performance of probabilistic selective cracking as described in Section 3.3.6. Sel20%, Sel50%, Sel80% probabilistically performs stochastic cracking 20%, 50%, 80% of the time respectively. The performance of the selective cracking is on par with the progressive cracking. Both show promising results to further reduce the initialization cost and quickly adapt in the pathological workload.

Naive Approaches

A natural question is why we do not simply impose random queries to deal with robustness. The next experiment studies such approaches using the same set-up

as before with the sequential workload.

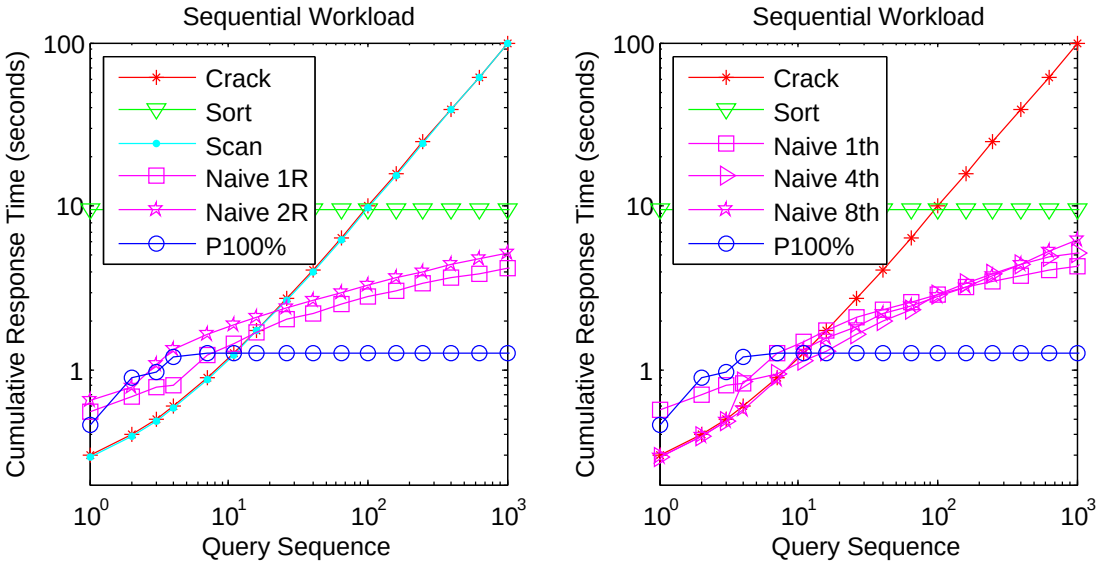


Figure 3.10: Simple cases

In the alternatives shown in Figure 3.10, Naive 1R and Naive 2R forces 1 and 2 random queries respectively for every user query. The less aggressive naive approaches: Naive 1th, Naive 4th, and Naive 8th force 1 random query every 1, 4, and 8 user queries. Notably, all these approaches improve over original cracking by one order of magnitude in cumulative cost. However, Stochastic Cracking (P100% or any other Stochastic Cracking variants in Figure 3.9) gains another order of magnitude, as it integrates its stochastic cracking actions *within* its query-answering tasks. This rationale is the same as that in original cracking: physical refinement is not an "afterthought", an action merely triggered by a query; it is *integrated* in the query processing operators and occurs on the fly. Furthermore, Stochastic Cracking quickly converges to low response times (its curve becomes flat), while naive approaches do not converge even after 10^3 queries.

3.4.2 Stochastic Cracking under Random Workload

We have now shown that Stochastic Cracking manages to improve over plain cracking with the sequential workload. Still, it remains to be seen whether it maintains the original cracking properties under a random workload as well.

Figure 3.11 repeats the experiment of Section 3.1.1 for the random workload, but adds Stochastic Cracking in the picture. The performance of plain cracking and Sort is as in Section 3.1.1; while Sort has a high initialization cost, plain cracking improves in an adaptive way and converges to low response times. Figure 3.11 shows that *all* our Stochastic Cracking algorithms achieve a performance similar to that of original cracking, maintaining its adaptability and good proper-

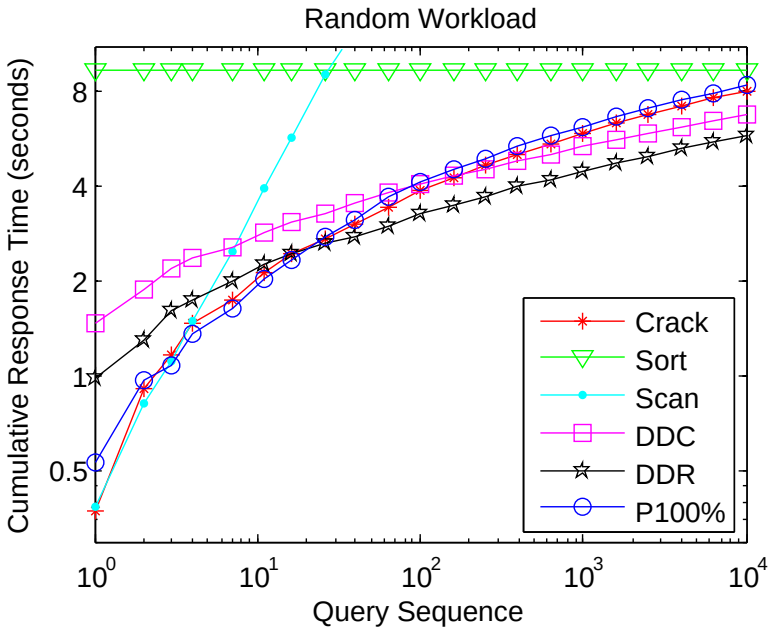


Figure 3.11: Stochastic Cracking under Random Workload

ties regarding initialization cost and convergence. Moreover, the more lightweight progressive Stochastic Cracking alternative approaches the performance of original cracking quite evenly. Original cracking is marginally faster during the initialization period, i.e., during the first few queries, when the auxiliary actions of Stochastic Cracking operate on larger data pieces, hence are more visible. However, this gain is marginal; with efficient integration of progressive stochastic and query-driven actions, we achieve the same adaptive behavior as with original cracking. DDC and DDR manage to perform better than the original cracking due to more cracker indexes (pieces) introduced per query, thus they have faster convergence and better total cumulative response time in the long run.

3.4.3 Stochastic Cracking under Various Workloads

Table 3.2 presents the relative cumulative time from the best algorithm to run 10^4 queries under various workloads. The value 1.00 represents the best cumulative time with ratio one. The other values bigger than 1.00 represent the amount of factor slower than the best. The values enclosed by a square bracket denote the worst ratio. The column Det50% represents the selective cracking that deterministically perform stochastic cracking every two queries (explained in Section 3.3.6). In addition to individual workload patterns, Table 3.2 also depicts results for a Mixed workload representing a mixture of all workloads studied so far; it randomly switches between each workload in every 1000 queries.

We observe that Stochastic Cracking (DDR, DD1R, P100%) maintains its

	Cracking strategy (relative ratio to the best strategy)								
Workload	*Crack*	*Sort*	*DDR*	*DD1R*	*P100%*	*Sel20%*	*Sel50%*	*Sel80%*	*Det50%*
Random	1.41	[1.64]	1.02	1.00	1.47	1.40	1.43	1.54	1.43
Sequential	[1051.44]	10.43	1.00	1.31	1.45	1.75	1.70	1.55	1.80
SeqInv	[3027.23]	18.44	1.17	1.43	1.00	6.38	2.85	1.12	1.10
SeqRand	1.56	[1.77]	1.02	1.00	1.39	1.36	1.43	1.46	1.31
SeqNoOver	[1055.27]	10.47	1.00	1.32	1.46	1.71	1.73	1.53	1.79
SeqAlt	[1214.41]	9.18	1.02	1.00	1.51	3.85	1.56	1.51	14.73
ConsRandom	1.17	[2.10]	1.00	1.01	1.73	1.33	1.45	1.68	1.43
ZoomIn	[224.49]	6.84	1.12	1.00	2.72	1.17	1.08	2.92	1.19
ZoomOut	[843.00]	8.41	1.13	1.04	1.61	2.42	1.38	1.87	1.00
SeqZoomIn	2.31	[8.38]	1.00	1.23	1.38	1.59	1.18	1.55	2.13
SeqZoomOut	[805.84]	11.62	1.25	1.65	1.78	1.00	1.44	1.82	1.95
Skew	1.04	[1.74]	1.07	1.00	1.60	1.25	1.40	1.62	1.33
ZoomOutAlt	[517.66]	7.83	1.00	1.06	1.42	2.20	1.35	1.63	208.68
SkewZOA	[1538.13]	18.81	1.16	1.57	1.00	4.89	2.53	1.08	579.09
Periodic	[4.38]	2.62	1.00	1.02	1.54	1.72	1.52	1.49	2.14
Mixed	[30.10]	3.92	1.04	1.00	1.65	1.53	1.49	1.56	4.19
SkyServer	[62.32]	2.18	1.00	1.06	1.45	2.99	1.91	1.59	1.71

Table 3.2: Various workloads

robust behavior across various workloads. On the other hand, original cracking fails significantly with most of them, being two or more orders of magnitude slower than Stochastic Cracking. Sort becomes the worst where original cracking performs better. Original cracking behaves well only for the workloads that contain enough random elements by themselves. The table shows the most robust stochastic cracking variants under various workloads are DDR and DD1R.

Comparing Stochastic Cracking with its selective variants, we observe that the deterministic selective cracking Det50% behaves rather well in many scenarios, but still fails in some of them, i.e., it is not robust. This is due to the fact that it follows a query-driven logic with every second query; thus, it is vulnerable to patterns that happen to create big column pieces during (some of the) odd queries. On the other hand, the probabilistic selective cracking strategies (Sel20%, Sel50%, Sel80%) provides an overall robust solution, i.e., it does not fail in any of the workloads. By randomizing the decision on whether to apply Stochastic Cracking or not for every query, it avoids the deterministic bad access patterns that may appear with each workload. In the SkyServer workload, selective cracking can be up to two orders of magnitude slower than the pure Stochastic Cracking. This is due to the fact that selective cracking may fall into bad access patterns (even if only a few), as it eschews stochastic operations where it should not. None of the Selective Stochastic Cracking variants manage to present an overall better performance than pure Stochastic Cracking.

Figure 3.12 shows a more detailed per query cumulative response time of the various workloads from Figure 3.4. We only show the DD1R variant to avoid clutter (DDR and MDD1R perform similarly with DD1R). Stochastic cracking performs robustly across the whole spectrum of workloads. On the other hand, original cracking fails in many cases; in half of the workloads, it loses the low

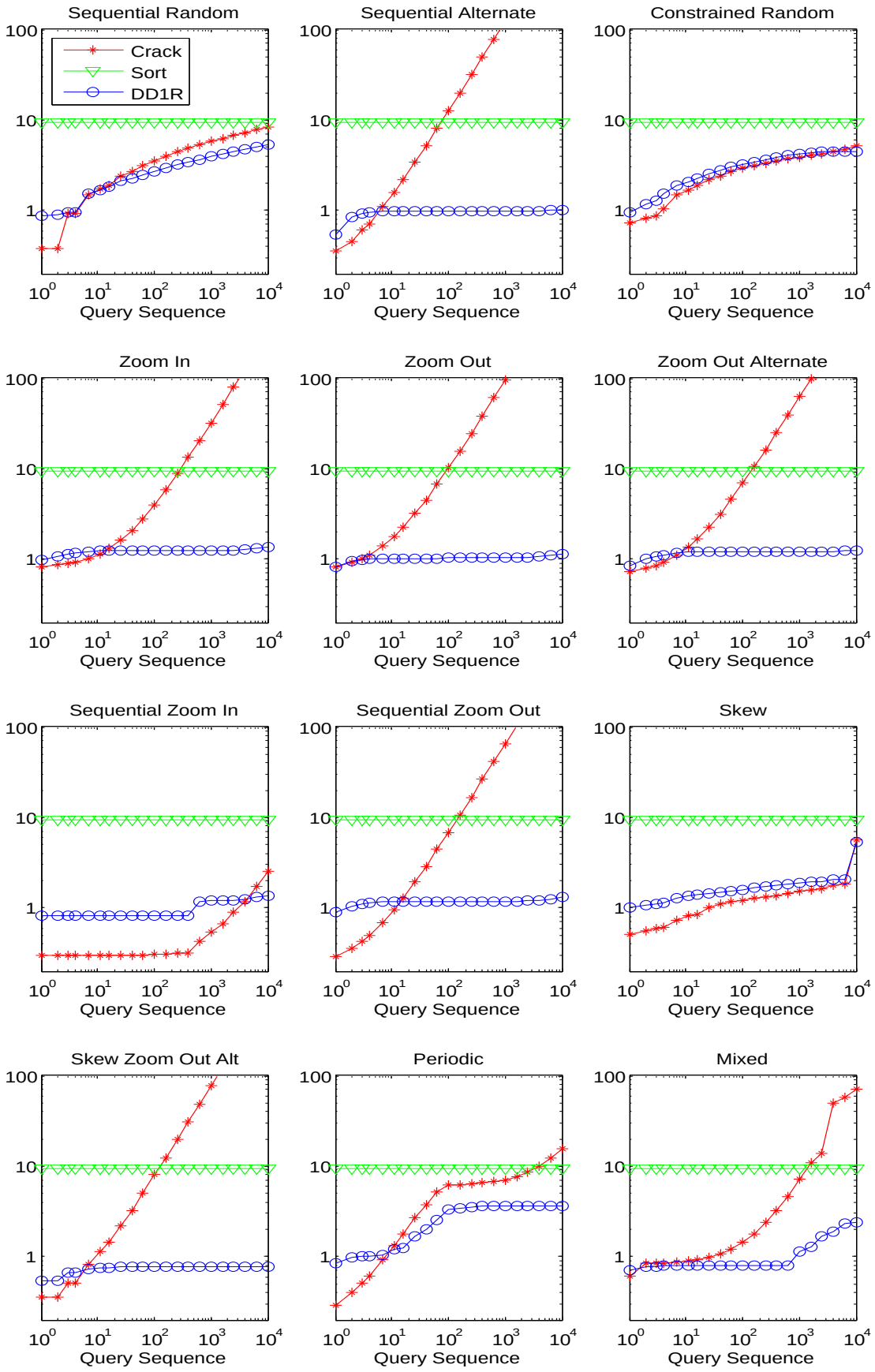


Figure 3.12: Various workloads under Stochastic Cracking

initialization advantage over full indexing, and performs significantly worse than both Stochastic Cracking and full indexing over the complete workload. For those workloads where original cracking does not fail, Stochastic Cracking follows a similar behavior and performance.

3.4.4 Stochastic Cracking under Varying Selectivity

Table 3.3 shows how Stochastic Cracking maintains its workload robustness with varying selectivity. It shows the cumulative time (seconds) required to run 10^4 queries. Stochastic cracking maintains its advantage for all selectivity with the sequential workload. We observe that DD1R achieves better cumulative times, while progressive Stochastic Cracking sacrifices a bit more in terms of cumulative costs to allow for a smaller individual query load at the beginning of a workload query sequence (see also Figure 3.9). Furthermore, higher selectivity factors cause Scan and progressive cracking to increase their costs, as they have to materialize larger results (whereas the other strategies return non-materialized views as they collect all result tuples in a contiguous area). For progressive cracking, that is only a slight extra cost, as it only has to materialize tuples from the array pieces (at most two) not fully contained within a query's range.

	Random Workload selectivity %						Sequential Workload selectivity%					
Algor.	10^{-7}	10^{-5}	10^{-2}	10	*50*	*Rand*	10^{-7}	10^{-5}	10^{-2}	10	*50*	*Rand*
Scan	2886	2881	3017	3864	4658	4163	957	957	1108	2058	4360	3187
Sort	9.3	9.3	9.3	9.3	9.3	9.3	9.3	9.3	9.3	9.3	9.3	9.3
Crack	6.3	6.3	5.8	5.8	5.9	5.8	943	652	652	652	653	6.0
DD1R	5.6	5.7	5.8	5.8	5.8	5.8	1.1	1.2	1.3	1.3	1.3	5.5
P10%	9.3	9.4	10.3	10.3	10.8	11.0	1.5	1.5	1.6	2.8	2.7	9.5

Table 3.3: Varying selectivity

3.4.5 Adaptive Indexing Hybrids

In recent work, cracking was extended with a partition/merge logic [57]. Therewith, a column is split into multiple pieces and each piece is cracked independently. Then, the relevant data for a query is merged out of all pieces.

These partition/merge-like algorithms improve over original cracking by allowing for better access patterns. However, as they are still based on what we call the blinkered query-driven philosophy of original cracking, they are also expected to suffer from the kind of workload robustness problems that we have observed. Figure 3.13(left) demonstrates our claim, using the sequential workload. We use

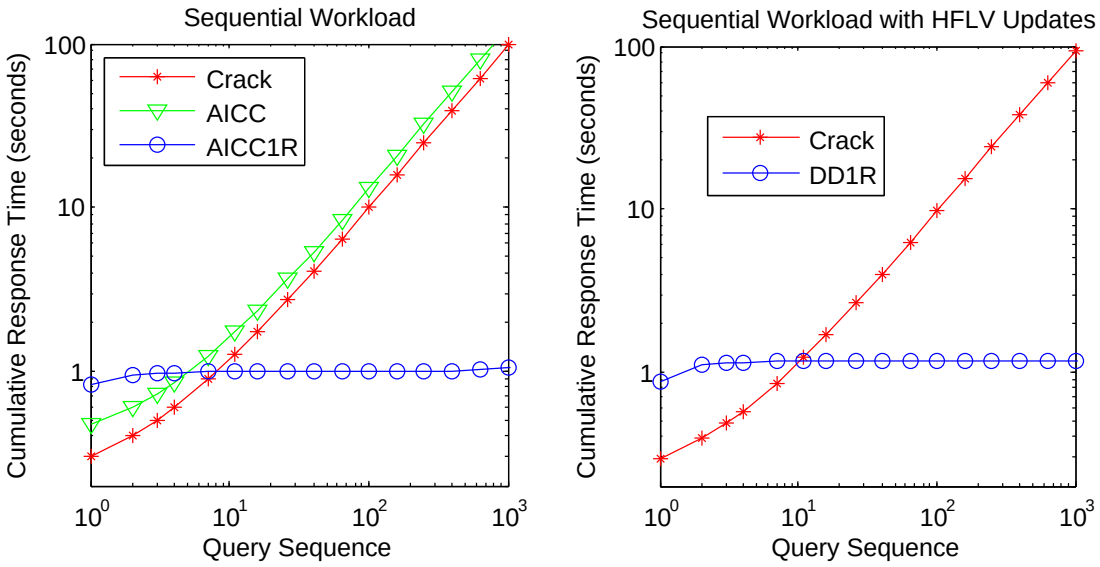


Figure 3.13: Stochastic Hybrids

the Hybrid Crack-Crack (AICC) method from [57]. It fails to improve on its performance, as it blindly follow the workload. Besides, due to the extra merging overhead imposed by the sequential workload, AICC is slightly slower than original cracking. In order to see the effect and application of Stochastic Cracking in this case as well, we implemented the basic stochastic cracking logic inside AICC, in the same way we did for DD1R. The same figure, above, shows the performance of AICC1R, namely our algorithm, which, in addition to the cracking and partition/merge logic, also incorporates DD1R-like stochastic cracking in one go during query processing. Our stochastic AICC variant gracefully adapts to the Sequential Workload, quickly converging to low response times. Thereby, we demonstrate that the concept of stochastic cracking is directly applicable and useful to the core cracking routines, wherever these may be used.

3.4.6 Stochastic Cracking under Updates

Figure 3.13(right) shows the Stochastic Cracking performance under updates. Given that stochastic cracking maintains the core cracking architecture, the update techniques proposed in [55] apply here as well. Updates are marked and collected as *pending* updates upon arrival and will only be merged to the cracker column when a query request values that intersect with the values in the pending updates. The presence of updates do not help in bringing robustness to the original cracking as they have no influence in refining the physical data store. The figure presents the performance with the Sequential workload when updates interleave with queries. We test a high-frequency low-volume (HFLV) update

scenario where 10 random updates arrive every 10 queries. Stochastic Cracking maintains its robust behavior, while the original cracking still fails to adapt. We obtained the same behavior with varying update frequency (as in [55]).

3.4.7 Stochastic Cracking under Real Workloads

In our next experiment, we test Stochastic Cracking under the SkyServer workload [64]. The SkyServer contains data from the astronomy domain and provides public database access to individual users and institutions. We used a 4 Terabyte SkyServer data set. To focus on the effect of the select operator, which matters for Stochastic Cracking, we filtered the selection predicates from queries and applied them in exactly the same chronological order in which they were posed in the system. Figure 3.14(b) depicts the exact workload pattern logged in the SkyServer for queries using the "right ascension" attribute of the "Photoobjall" table. The Photoobjall table contains 500 million tuples, and is one of the most commonly used ones. Overall, we observe that all users/institutions pose queries following non-random patterns. The queries focus in a specific area of the sky before moving on to a different area; the pattern combines features of the synthetic workloads discussed in Section 3.4.3. As with those workloads, here too, the fact that queries focus on one area at a time creates large unindexed areas. Figure 3.14(a) shows that plain cracking fails to provide robustness in this case as well, while Stochastic Cracking maintains robust performance throughout the query sequence; it answers all 160 thousand queries in only 21 seconds, while original cracking needs more than 1000 seconds. When the stochastic cracking finished answering all the queries, a full indexing approach (Sort) is not even halfway preparing its physical data store and has not yet answered a single query. Sort needs 70 seconds to answer all the queries which is three times more than the stochastic cracking, while a plain scan more than 8000 seconds. These results establish the advantage of Stochastic Cracking with a real workload.

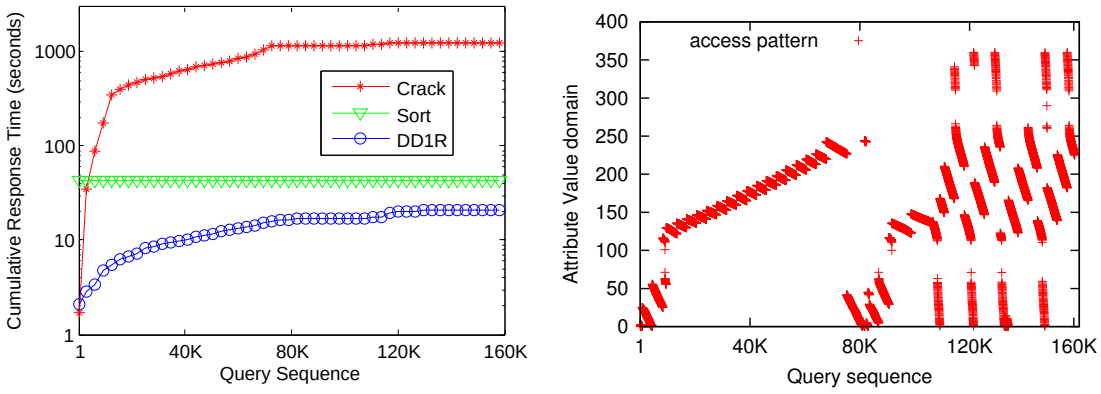


Figure 3.14: Cracking on the SkyServer Workload

3.5 Conclusion

We have witnessed the importance of investing a small computation to give an overall robust algorithm under dynamic environments while maintaining the original algorithm efficiency. Stochastic cracking spends an effort in observing the underlying physical data store (i.e., the piece size) and perform stochastic crack(s) with minimal overhead. Stochastic cracking clearly improves over original cracking by being robust in workload changes while maintaining sub-linear algorithm complexity and all original cracking features when it comes to adaptation. Furthermore, we have established that, given the unpredictability of dynamic workloads, there is no "royal road" to workload robustness, i.e., no easy way out of the necessity to apply stochastic cracking operations with *every single query*.

Chapter 4

Large Graph Processing

Large graphs naturally arise from the web, social networks, biology, etc. These graph instances are unique, have many special properties which are hard to synthesize, and have intrinsic values that fascinate researchers to analyze them deeper. Unfortunately, these graphs can easily reach hundreds of gigabytes in size with hundred millions of vertices and billions of edges. Analyzing such graphs is a big data problem. Consider the Facebook social network graph. Currently, it has more than 800 million users (and growing) with an average user has 130 friends [4]. This means that a plain adjacency-list graph data structure alone would need at least $800 \times 10^6 \times 130 \times 8$ bytes $\approx$ 832 GB space. Additional data structures will increase the space requirement further. Processing/analyzing such large graphs becomes challenging as the computation, storage, and memory required exceeds the capacity of a single machine. Classical sequential algorithms that are used to process such graphs simply fail because the input dataset is too large to fit into the main memory. The recent *Graph 500* ranking for evaluating supercomputers to complement the *Top 500* using data-intensive graph problems including social networks [5, 6] is also indicative of the importance of processing large graphs as HPC workloads.

While it is possible to use a very large and expensive machine with TB-sized memory, currently it is more economical and practical to use a cluster of commodity machines [1] or use cloud computing. This requires the graph to be partitioned and distributed to a cluster of machines to be processed in parallel. This naturally requires dealing with distributed computing issues (e.g. data partitioning and distribution, load balancing, scheduling, fault tolerance, communication, etc) which are not trivial and difficult to be realized without a well established framework. Recently, the MapReduce (MR) [25] framework was introduced by Google to abstract the above issues away from the user. MR provides a simple programming model and quickly becomes de facto standard for processing very large datasets over thousands of commodity machines. Soon after,

Hadoop, an open source implementation of MR framework emerged [7] which is widely used by companies to process large scale data such as Facebook, Amazon, Yahoo, etc. Adopting a new platform such as MR entails retrofitting/reinventing existing algorithms to comply with the constraints of the new platform. The most difficult challenge in developing an algorithm in the MR framework is that the algorithm must be written in the form of a MAP function and a REDUCE function. Both of which have to work in local and stateless manner. This means that sequential algorithms which make use of global state may not be so easily turned into its MR counterpart. Moreover, the performance cost metric of operating in MR platform is different than the traditional complexity metric. One has to take into account the I/O and communication overheads which are not captured in the usual algorithm complexity analysis. In practice, they can dominate the computation. With the large scale of the graph to be processed and the limited choice of the available framework, the questions that can be asked to the graph become limited.

Among many graph problems, we are interested in those that have solutions with quadratic (or more) runtime complexity. In this chapter, we look at how we can redesign classical graph algorithms, in particular the Maximum-Flow (max-flow) algorithms, which has quadratic runtime complexity into practical MapReduce-based algorithms that perform in linearly in practice. Existing max-flow algorithms (for integer capacity flow) have about quadratic (or more) runtime complexity [35]. The challenge is whether real-world graphs with $\sim 10^9$ vertices and $\sim 10^{11}$ edges (or more) can be effectively processed.

Fortunately, many real-world graphs (such as social networks, the World Wide Web, Wikipedia, etc.) inherently have small-world properties [13, 86, 12, 100], e.g. the graphs have small diameter. That is, the length of the shortest-path between any two vertices in the graph is expected to be small [85]. The small diameter property allows us to design new max-flow algorithms that scale linearly in practice in terms of graph size and its diameter. We develop and evaluate new MapReduce algorithms based on two well known existing max-flow algorithms, the Push-Relabel algorithm and the Ford-Fulkerson method, on their efficiency in processing very large real-world social network graphs using the MR framework. We discovered that while the Push-Relabel algorithm works in local manner (i.e., each vertex only requires information about its neighbors), it is not suitable in the MR settings due to the arbitrary large number of *MR rounds* needed. On the contrary, we found that algorithms based on the Ford-Fulkerson method have a large potential for parallelization. By utilizing the high potential for parallelization of the Ford-Fulkerson method and exploiting the small-world property, we design and develop a new and practical MapReduce-based Maximum-Flow algorithm

for processing large small-world network graphs / social networks. We are able to process very large real-world graphs with 411 million vertices and 31 billion edges using a cluster of 21 machines. We show that our algorithm is scalable in terms of the max-flow value, the graph size, and the number of machines. We also devised novel algorithm-system optimizations which improve the performance of the initial design by a factor of up to 14x (see Sec. 4.5). We believe that the techniques presented in this chapter can give new insights for scaling other graph related algorithms.

Our work on the techniques in parallelizing the Ford-Fulkerson method that take advantage of the small diameter property of the graph, its design, implementation, and evaluation on top of the MapReduce framework is published in [49].

4.1 Overview of the MapReduce Framework

The *MapReduce* (MR) framework was introduced by Google [25] as a simple programming model to run distributed computation on very large datasets using large clusters of commodity machines. MR requires the input dataset to be a series of (independent) *records* consisting of $\langle key, value \rangle$ pairs. Each record must be designed so that it can be processed in isolation, i.e., local to the record itself. This allows a very large number of (independent) records to be partitioned and distributed across machines and processed in parallel. The input records are stored in a distributed file system (DFS_{MR}) – Google's MR uses GFS [32] while Hadoop uses HDFS [99]. Our implementations use Hadoop, the open source implementation of MR.

The MR framework manages the nodes in a cluster. In Hadoop, one node is designated as the master node and the rest are the slave nodes. An MR job consists of the input records and the user's specified MAP and REDUCE function. When an MR job is submitted for execution, the master node schedules a number of map and reduce tasks. The slave nodes will spawn a number of workers to execute (in parallel) the tasks scheduled by the master node. Each input record will be processed by a worker, called a mapper, that applies the MAP function possibly outputting a number of *intermediate* records, also in the form of $\langle key, value \rangle$ pairs. After all mappers are finished, the reduce phase begins. Intermediate records having the same *key* are grouped together (which may require sending/shuffling the intermediate records between workers in different slave nodes). Each group of intermediate records having the same *key* is then processed by a worker, called a reducer, that applies the REDUCE function possibly outputting a number of records which are the final result of the MR job. The

MAP and REDUCE functions need to be *stateless* in the MR framework as they only take the input record(s) and produce (intermediate) records.

Executing an MR job incurs large overheads as the main operations involve reading and writing to the DFS_{MR} and shuffling data between nodes – resulting in large amounts of disk I/O and network traffic, roughly proportional to the data size. For large enough data, the cost of fetching and shuffling the data may be much larger than the computation cost of executing the user's MAP and REDUCE functions. Hence, optimizations to reduce MR overheads are necessary.

MR allows the total size of the input to be far larger than the total available memory of the slave nodes in the cluster. Each slave node can run a number of mappers and reducers concurrently. However, the number of workers is limited by the number of processors and memory in the node as well as the memory requirement of a worker when processing records. In MR, the memory requirement to process one (or more) record(s) by a mapper (or reducer) is expected to be far smaller than the memory capacity of any node in the cluster. This allows workers in MR to process an arbitrarily (large) number of input records as long as the total memory requirement of the running workers is within the node's memory capacity.

MR provides some special aggregation operations which have some state such as counters. However, these counters are meant to be read after the MR job has finished. While MAP and REDUCE are meant to be stateless from the MR perspective, we shall see in Sec. 4.5.1 that a form of state can be effective – using an external stateful process which is contacted from inside the MAP or REDUCE function.

For simple uses of MR (see [25]), a single MR job is sufficient. However, in complex applications such as computing the diameter of a large graph [67] or computing max-flow (this thesis), several MR jobs are chained together – the output of the current MR job becomes the input of the next MR job. We refer a single MR job as an *MR round* and a chain of MR jobs as a *multi-round* MR.

Given the large overheads incurred as part of the execution of an MR job, together with the synchronization between rounds in a multi-round MR, we will argue for and show that an appropriate measure of the complexity of a multi-round MR is the number of rounds rather than more traditional algorithmic complexity measures. We also show that performance gain can be achieved by lowering the number of rounds and increasing the parallelism in each round, getting more work done in each round and avoid spilling the work to the next round.

Recently, Google proposed a new specialized framework for processing large-scale graphs based on a bulk synchronous parallel model, called Pregel [82] which

is proprietary and unavailable for use outside Google. The open source implementations, Apache Hama and Giraph, are still in development [2]. This limits our choice to MapReduce. Nevertheless, we believe that the ideas presented in this paper are also applicable to the bulk-synchronous parallel model.

4.2 Overview of the Maximum-Flow Problem

The maximum-flow (max-flow) problem, namely, to find the maximum-flow in a directed graph from source to sink given edge capacity constraints is a classic combinatorial optimization problem. In the context of Internet scale graphs, max-flow problems arise in problems such as discovering spam sites [94], community identification [29, 58], preventing Sybil attacks [27] in P2P networks [107], honest online content vote counting [105], etc. However, due to the rapid growth of online communities and social networks, the size of real-world graphs has grown far larger than the amount of available memory in conventional machines. Such large graphs create a big data problem on how to scale existing max-flow algorithms. Our objective is to effectively compute the max-flow for such Internet scale graphs, in particular, large social networks.

In the MR framework, some large graph algorithms have been developed such as s-t graph connectivity, MST [68], estimating the approximate graph diameter [67], social content matching [24], centrality [66], etc [9]. There also exist some design patterns such as in-mapper combining, schimmy, and graph partitioning [80] to improve MR performance. Optimizations that take into account the cluster's intra and inter-node bandwidth to partition the graph have been proposed [22]. In contrast, this thesis addresses the design and optimization issues in developing a complex MR algorithm. We give a novel way of transforming a sequential algorithm into a highly parallel MR algorithm using speculative execution and introduce new MR-optimizations: a stateful extension for MR and space versus time optimizations.

In this section, we will give a brief overview of the general maximum-flow problem and two well known maximum-flow algorithms, Push-Relabel algorithm and the Ford-Fulkerson method. In the next sections, we will elaborate in detail on our MapReduce-based design, implementation, and evaluation on both algorithms.

4.2.1 Problem Definition

A *flow network* $G = (V, E)$ is a directed graph where each edge $(u, v) \in E$ has a non-negative *capacity* $c(u, v) \geq 0$. There are two special vertices in a flow

network: the *source* vertex s and the *sink* vertex t. Without loss of generality, we can assume there is only one source and sink vertex, which we call s and t respectively. A *flow* is a function $f : V \times V \rightarrow \mathcal{R}$ satisfying the following three constraints:

- **Capacity Constraint**: $f(u, v) \leq c(u, v)$ for all $u, v \in V$,
- **Skew Symmetry**: $f(u, v) = -f(v, u)$ for all $u, v \in V$, and
- **Flow Conservation**: $\sum f(u, v) = 0$ for $u \in V - \{s, t\}$ and $v \in V$.

The flow value of the network is $\sum f(s, v)$ for all $v \in V$. In the *max-flow* problem, we want to find a flow f^* such that $|f^*|$ has maximum value over all such flows. Two important concepts used in flow networks are the following.

- **Residual Network**. For a given flow network $G = (V, E)$ with a flow f associated to it, the *residual network* $G_f = (V, E_f)$ is the set of edges E_f that have positive *residual capacity* c_f. That is, $E_f = \{(u, v) \in E : c_f(u, v) = c(u, v) - f(u, v) > 0\}$.
- **Augmenting Path**. An *augmenting path* is a simple path from s to t in the residual network.

There are two well known solutions to the max-flow problem namely the Push-Relabel algorithm [37] and Ford-Fulkerson method [30] .

4.2.2 The Push-Relabel Algorithm

Let the *height* and the *excess flow* of a vertex u be h_u and x_u. The height of s and t are fixed to $h_s = |V|$ and $h_t = 0$. The Push-Relabel algorithm uses the following two operations [37]:

- **Push**(u, v) is performed if $(u, v) \in E_f$ and v has height equal to $h_u - 1$. The excess flow x_u and $f(v, u)$ will be decreased by δ where $\delta = min(x_u, c_f(u, v))$ while x_v and $f(u, v)$ will be increased by δ,
- **Relabel**(u) is performed when all $v \in V, (u, v) \in E_f$ have height $h_u \leq h_v$. Relabel sets h_u to $1 + min\{h_v : (u, v) \in E_f\}$.

Initially, the excess flow of each vertex is 0 except for s which has infinite excess flow. A *preflow* is created by pushing the excess flow of s to all $v : (s, v) \in E_f$. Push or relabel operations can then be performed in any order for all $u \in V - \{s, t\}$ and $x_u > 0$ until no more push nor relabel operations can be performed in which case the maximum flow is the amount of excess flow arriving at t. Although push

and relabel can be performed in any order, the ordering impacts performance. In practice, the distance and gap relabeling heuristics [23] can make a significant difference. The Push-Relabel algorithm works in a localized manner, that is, when processing a vertex, it only requires the information of the vertex and its neighbors, making the algorithm suitable for distributed processing. Since there is no dependency between push and relabel operations, the Push-Relabel algorithm can be executed in parallel. The operations can be applied arbitrarily as long as the pre-conditions hold.

Parallel Push-Relabel max-flow implementations have been developed for SMP architectures [14]. There is also a parallel max-flow algorithm without locks [98] but it is for a PRAM-like computation model which is not practical in a cluster/cloud setting. Classical max-flow algorithms [35] require the entire graph to be fit in memory. In contrast, our goal is to compute max-flow on real small-world graphs (rather than arbitrary graphs) which are far larger than the available machine memory using the MR framework. We develop a new MR algorithm based on the Push-Relabel algorithm and evaluate its performance in Section 4.3.

4.2.3 The Ford-Fulkerson Method

The idea of the Ford-Fulkerson method is to find an augmenting path p. The flow value of G is then increased by augmenting the flow along path p. This is then repeated until no more augmenting paths can be found, in which case, the max-flow is obtained [30]. A basic implementation of this method runs in $O(|f^*|E)$, that is a depth-first search is run $|f^*|$ times to find $|f^*|$ augmenting paths. Major improvements have been discovered in the past decades based on the Ford-Fulkerson method [36]. The Edmonds-Karp algorithm [28] implements the Ford-Fulkerson method by always augmenting the shortest augmenting path in the residual network, giving a running time of $O(VE^2)$, while Dinic's algorithm implements using the level graph and blocking flow which run in $O(V^2E)$ [26]. It turns out that we can design highly parallelizable algorithms based on the Ford-Fulkerson method. We present our design, implementation and evaluation of our new MR max-flow algorithms based on the Ford-Fulkerson method in Section 4.4.

4.2.4 The Target Social Network

Our focus is on computing Max-Flow using MR on large real-world graphs which represent graphs on the online communities, the world-wide-web, social networks, etc. One important property of such graphs is the small world properties, in particular, having a small average diameter. We chose the Facebook social network for our experiments because it is perhaps the largest real social network avail-

able and the graph has been shown to have a very small average diameter (e.g, 4.7) [13]. We crawled Facebook using a strategy which creates small-world-like subgraphs [96]. The crawler algorithm crawls the next user that has the most number of connections to the currently crawled users. We crawled in stages and created a checkpoint when the graph grows into certain sizes then we continue crawling. This way, we created several versions of the graph with different sizes where the smaller graph is the subset of the larger graph. We store the crawled graph into several subsets, FB1 to FB6, where FBi is a subgraph of FBj for $i < j$. Based on the number of Facebook users and average degree [4], we estimate that the FB6 graph is about half the number of users of the full Facebook network.

Graph	**Vertices**	**Edges**	**Size**
FB0	5 M	52 M	157 MB
FB1	21 M	112 M	587 MB
FB2	73 M	1,047 M	6 GB
FB3	97 M	2,059 M	13 GB
FB4	151 M	4,390 M	30 GB
FB5	225 M	10,121 M	69 GB
FB6	411 M	31,239 M	238 GB

Table 4.1: Facebook Sub-Graphs

Table 4.1 shows the 6 sub-graphs. The *Size* column gives the size of the graph as it is stored in HDFS in SequenceFile format as a list of vertices with the data structure described in Sec. 4.4.3. During the execution, the size of the graph may expand as more information is stored in a vertex.

Note that Facebook sets a limit of 5000 friends. If a vertex has too many edges, without loss of generality, it can be decomposed into several vertices of smaller degree to maintain the maximum degree of a vertex. Thus, throughout our MR-based algorithm described in this chapter, we assume each vertex in the graph has degree at most $C = 5000$. In case a vertex has X edges where $X > C$ edges, the vertex can be decomposed into a complete graph of $Y = \lceil X/C \rceil$ vertices and edges with infinite capacities connecting them. The original edges attached to X are then distributed evenly among the Y vertices. This process can be repeated until all the vertices has at most C edges. Each transformation may cause the diameter to enlarge by one, however, it will not change the max-flow value. Imposing the graph structure to have at most C edges is useful because each MAP (and REDUCE) function will have to load the vertex and its edges into machine's memory. It may be the case the vertex has so many edges that it can exceed memory capacity. As such, this transformation is important for MR algorithms to keep the maximum record size small to maintain low memory requirement for the workers (see Section 4.1).

4.3 MapReduce-based Push-Relabel Algorithm

As described in Section 4.2.2, the Push-Relabel algorithm seemed to fit in the MR framework as it is able to work in local manner and able to be executed in parallel. In this section we develop a new MapReduce-based Push-Relabel algorithm which we call the PR_{MR} algorithm. In particular, we have to express the Push-Relabel algorithm in terms of stateless MAP and REDUCE functions to operate on a graph which is represented by a set of records of $\langle key, value \rangle$ pair. Before we jump to the core of the MAP and REDUCE function, we first describe how we represent a graph in PR_{MR}.

4.3.1 Graph Data Structures for the PR_{MR} Algorithm

In MR, a graph is represented as a series of records. We assume that every vertex in the graph is represented by an unique identifier (ID) which can be used as a key. Thus, we model each vertex u and its edges as a $\langle key, value \rangle$ pair where the *key* is the vertex ID of u and the *value* is the vertex internal data structure containing all the information about vertex u of the form $\langle h_u, x_u, E_u \rangle$ where h_u and e_u are the height and excess flow respectively of vertex u (see Section 4.2.2). The edges of u, E_u, is a list of tuples where each tuple represents an edge which consists of $\langle e_v, e_h, e_f, e_c, e_d, e_m \rangle$. e_v is the ID of the neighboring vertex of the edge. e_h is the height of the neighboring vertex e_v. e_f is the value of the flow from u to e_v. e_c is the capacity of the edge. e_d is the distance of vertex e_v to t. e_m is the distance's timestamp of e_d. We define e_r to be the edge residue where $e_r = e_c - e_f$. We note that some of the values in E_u are redundant to ensure that a vertex can operate on local information alone, e.g. the height of a vertex u is stored in vertex u and also in each of u's neighbors' edge to u.

4.3.2 The PR_{MR} MAP Function

To compute the max-flow, the PR_{MR} algorithm requires several rounds of MR jobs. In each round of a MR job, the map phase applies the PR_{MR}'s MAP function in parallel for each record from the previous round (or from original input dataset, if this is the first round). The MAP function produces a set of intermediate records which are then shuffled across machines, grouped by vertex ID and applied the REDUCE function. The output of the REDUCE function is written to the distributed file system (DFS_{MR}) to be used in the next round of PR_{MR}.

Figure 4.1 and Figure 4.2 give the MAP and REDUCE functions for PR_{MR}. The EMIT-INTERMEDIATE operation is used in the MAP function for emitting intermediate records and the EMIT is used in the REDUCE function for emitting

function MAP$_{PR}(u, \langle h_u, x_u, E_u \rangle)$
1. **if** (**round** $= 0$ and $u = s$)
2. $\quad h_u = \mathbf{N}$ // *initial height for source* $= |V|$
3. $\quad$ **foreach** $(e \in E_u)$ **do**
4. $\quad\quad e_f = e_f + e_r$ // *Pre-Flow Push*
5. $\quad\quad$ EMIT-INTERMEDIATE$(e_v, \langle 0, e_r, \langle \langle u, h_u, -e_r, 0, 0, 0 \rangle \rangle \rangle)$
6. **if** $(u = t)$ // *Generate New Distance Label*
7. $\quad$ **foreach** $(e \in E_u : e_f > -e_c)$ **do**
8. $\quad\quad$ EMIT-INTERMEDIATE$(e_v, \langle 0, 0, \langle \langle u, 0, 0, 0, 0, round + 1 \rangle \rangle \rangle)$
9. **else** // *Propagate Distance Label*
10. $\quad m_u = \{max(e_m) | e \in E_u\}$ // *u's timestamp*
11. $\quad d_u = \{min(e_d) + 1 | e \in E_u : e_m = m_u\}$
12. $\quad$ **foreach** $(e \in E_u : e_f > -e_c)$ **do**
13. $\quad\quad$ EMIT-INTERMEDIATE$(e_v, \langle 0, 0, \langle \langle u, 0, 0, 0, d_u, m_u \rangle \rangle \rangle)$
14. **while** $(u \neq s$ and $u \neq t$ and $x_u > 0)$ // *Discharge Vertex u*
15. $\quad$ INCR('discharge_count')
16. $\quad E_s =$ sort E_u by $e_v \downarrow, e_d \uparrow, e_h \uparrow, e_r \downarrow$
17. $\quad$ **foreach** $(e \in E_s : e_r > 0,\ h_u > e_h,\ x_u > 0)$ **do**
18. $\quad\quad$ // *Push Operation*
19. $\quad\quad \delta = \min(x_u, e_r)$
20. $\quad\quad x_u = x_u - \delta;\ f_i = f_i + \delta$
21. $\quad\quad$ EMIT-INTERMEDIATE$(e_v, \langle 0, \delta, \langle \langle u, 0, -\delta, 0, 0, 0 \rangle \rangle \rangle)$
22. $\quad$ **if** $(x_u > 0)$ // *Relabel Operation*
23. $\quad\quad new_h = 1 + \{min(e_h) | e \in E_u, e_r > 0\}$
24. $\quad\quad \Delta h = new_h - h_u;\ h_u = new_h$
25. $\quad\quad$ **foreach** $(e \in E_u)$ **do** // *notify* Δh *to neighbors*
26. $\quad\quad\quad$ EMIT-INTERMEDIATE$(e_v, \langle 0, 0, \langle \langle u, \Delta h, 0, 0, 0, 0 \rangle \rangle \rangle)$
27. EMIT-INTERMEDIATE$(u, \langle h_u, x_u, E_u \rangle)$

Figure 4.1: The PR$_{MR}$'s MAP Function

the final output records for the current MR job. The variables s, t, N, and *round* (which are the source s, sink t, number of vertices $|V|$, and the round number of the current MR job) are given as read-only parameters to the MAP and REDUCE functions. MR provides distributed event counters and operation INCR(c) means to increment counter c. However, these counters are only accurate to the master node after the current MR job finishes. We use the notation $g : i$ to denote line i in function g. The MAP$_{PR}$ function does three tasks (see the pseudocode in Figure 4.1):

- Initializes the source vertex's height and push a pre-flow in the first round (MAP$_{PR}$:1-5);
- Updates the distance label heuristics of each vertex and propagate to its neighbors (MAP$_{PR}$:6-13); and

- Discharges the excess for all vertices except s and t by pushing its excess to the neighbors and relabels its height as necessary (MAP$_{PR}$:14-26).

In the first round of MR, the source vertex height is set to N (MAP$_{PR}$:2) and a pre-flow will be pushed from s to its neighbors (MAP$_{PR}$:3-5). Since each vertex (tuple) works in isolation, the neighbors of s have to be notified about the s's height and the increase of excess of the pre-flow. This is done by outputting an intermediate tuple with s's height delta increase and excess delta increase of the neighboring vertex as well as the reverse flow (MAP$_{PR}$:5).

We use a distance labelling heuristic to reorder the priority of the push operation to guide the excess flow in a vertex to be pushed towards the sink vertex t [23]. Without such a good heuristic, the Push-Relabel algorithm will have poor performance. Since we cannot have global information about the distance, each vertex has its own copy of the distance label as well as its neighbors' distance label. Each round, the residual network may change whenever there are changes in edges' flow, causing some distance label to become invalid (saturated). We associate the distance label of each vertex with a timestamp to distinguish which distance label is the latest. We use the current MR round number as the timestamp.

The distance label and the timestamp are continuously updated and propagated to stay up-to-date (MAP$_{PR}$:6-13). Each round, the sink vertex t disseminates to its neighbors new distance labels with a larger timestamp (MAP$_{PR}$:6-8). The rest of the vertices can calculate their own distance label by selecting the largest timestamp from its neighbor (MAP$_{PR}$:10). The distance of vertex u is equal to 1 + the smallest distance label with the newest timestamp (MAP$_{PR}$:11). The newest timestamp and distance label of vertex u is then propagated to its neighbors (MAP$_{PR}$:12-13).

The push or relabel operation can be applied in any order for vertices other than s and t with positive excess. If vertex u satisfies the condition above, then it is possible to exhaust all of u's excess using a sequence of push and relabel operation on vertex u. This process is called **discharge** (MAP$_{PR}$:14-26). An event counter, 'discharge_count', is incremented whenever the discharge subroutine is executed (MAP$_{PR}$:15). The discharge process first reorders the edge list by decreasing timestamp, increasing distance label, increasing height, and decreasing edge residue (MAP$_{PR}$:16). Discharge then pushes excess x_u of the current vertex u to its neighbors until it is exhausted (MAP$_{PR}$:17-21). The push operation employs priorities based on the sorted edge list. It first pushes the excess to an edge that has the newest timestamp with the smallest distance label, i.e., closest to the sink. A push operation (MAP$_{PR}$:19-21) to an edge e first calculates the maximum excess flow δ that can be pushed through the edge. This flow amount δ is subtracted

from the excess x_u and added to the edge's flow (MAP$_{PR}$:20). A notification is sent to neighboring vertex e_v to increase its excess by δ and decrease its edge flow by δ (MAP$_{PR}$:21). The push operation will exhaust either all the edge residue e_r or the u's excess x_u.

If x_u is still positive after all the push operations, the relabel operation is performed (MAP$_{PR}$:22-26). Relabel first calculates the new height for u which is equal to 1 + the minimum height of the neighboring vertex where the connecting edge has positive residue (MAP$_{PR}$:23). The Δ_h increase in the height is recorded and the current height of u is updated (MAP$_{PR}$:24). All the neighboring vertices are then notified of the increase of vertex u's height (MAP$_{PR}$:25-26). Discharging continues until $x_u = 0$. Lastly, the vertex u and its value is emitted to carry it to the next round of MR (MAP$_{PR}$:27).

4.3.3 PR$_{MR}$ REDUCE Function

```
function REDUCE_PR(u, values)
1.  h_u = x_u = 0
2.  E_u = ⟨⟩
3.  foreach (⟨h_v, e_v, E_v⟩ ∈ values) do
4.      h_u = h_u + h_v
5.      x_u = x_u + x_v
6.      foreach (e ∈ E_v) do
7.          g = E_u.get(e_v)
8.          if (g ≠ ∅) // Merge edge g with e
9.              g_h = g_h + e_h
10.             g_f = g_f + e_f
11.             g_c = g_c + e_c
12.             if (g_m < e_m or (g_m = e_m and g_d > e_d))
13.                 g_m = e_m
14.                 g_d = e_d
15.         else
16.             E_u = E_u ∪ e
17. EMIT(u, ⟨h_u, x_u, E_u⟩)
```

Figure 4.2: The PR$_{MR}$'s REDUCE Function

The intermediate records emitted by the MAP function during the map phase will be grouped by its key (vertex ID) and each group which then consists of a list of *values* is applied the REDUCE function in the reduce phase. In the sense, the MAP function is responsible for disseminating messages/information (in the form of intermediate records) from one vertex to its neighbors and the REDUCE function is responsible to collect all the messages for each vertex and merge it to the vertex's main record that was emitted at (MAP$_{PR}$:27).

REDUCE$_{PR}$ takes in a key representing a vertex ID u, with a list of intermediate record values of vertices having the same vertex ID u. It then merges all those vertices into a single value for vertex u. First, it initializes the vertex u's height, excess, and edge list (REDUCE$_{PR}$:1-2). Then each vertex in the *values* is added/merged to the vertex u (REDUCE$_{PR}$:3-16). All the delta heights along with the original height for the vertex u are summed together (REDUCE$_{PR}$:4), so is its excess (REDUCE$_{PR}$:5). For the edges, an edge will be added to the current list of edge E_u if E_u doesn't contain e_v (REDUCE$_{PR}$:16), otherwise the edge will be merged with the existing edge (REDUCE$_{PR}$:9-14). Finally, the key and the merged values are emitted as the final output record for the current MR job (REDUCE$_{PR}$:17).

At the end of each MR round, the 'discharge_count' counter value is examined. If the value of this counter is zero, discharging was not possible, which means all the excess flow in the network has either gone to t or back to s. The max-flow algorithm then terminates.

4.3.4 Problems with PR$_{MR}$

Although the PR$_{MR}$ algorithm fits within MR, we discovered that the fit with MR is not as good as it initially appears. Firstly, the amount of parallelism decreases as more excess flows are admitted. This leads to only a few *active vertices* [77] in an MR job. Given the way MR works, it is not possible to selectively process only the active vertices. The entire graph still needs to be read, processed, and written back to the distributed file system, making PR$_{MR}$ inefficient. Secondly, an excess flow may be transferred from u to other vertices and get trapped (i.e., no residual edge to the sink from those vertices because of the residual network changes) and thus the excess flows need to flow back to u so that it can flow out again. In MR, each vertex can only apply a MAP function once per MR round which means one excess flow transfer requires one MR round. Hence, any trapping behavior in the residual network can lead to very high number of MR rounds.

Figure 4.3 illustrates a bad scenario for PR$_{MR}$. The graph initially starts with all vertices having zero height except s and each edge has capacity 1 and zero flow. The sink distance is zero and undefined for other vertices. In the first MR round, a pre-flow will be pushed with flow 1 from source to the neighbor. In round 2, the distance label has not yet propagated to the node with the excess flow. Thus the node holding the excess flow has no information on where to push the excess flow. If the excess is pushed upwards then it will be trapped. It will cause the excess to visit all the vertices on the upper part of the graph because the height restriction prevents pushing excess flow to the previous vertex which is now has higher height. In round 18, the excess can then be pushed downwards

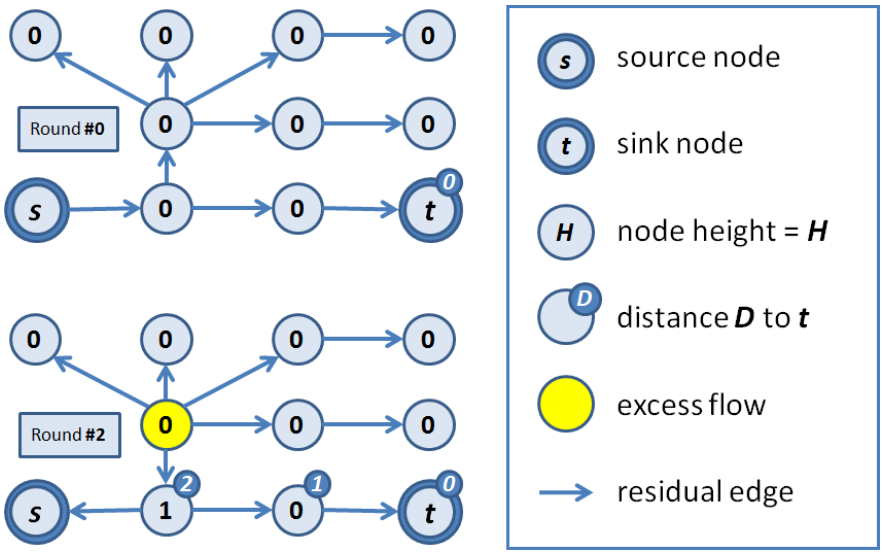


Figure 4.3: A Bad Scenario for PR_{MR}

back to the s's neighbor and then pushed rightwards to t completing in a total of 21 MR rounds. This example also illustrates the two problems of PR_{MR}. It has very low parallelism as there is only one excess flow in the graph. Each MR job only processes a small fraction of the graph and the rest of the graph is processed without giving any contribution towards task completion. A wrong excess flow leads to $O(|V|)$ MR rounds which is not practical given that a single MR job is costly in terms of I/O and network resources.

4.3.5 PR2_{MR}: Relaxing the PR_{MR}

As shown earlier, although Push-Relabel algorithm fits with MR in terms of stateless processing, it can be inefficient. The source of inefficiency seems to come from the height constraint for the push operation. This prevents the push operation to push backwards until all the unvisited vertices in the forward directions are tried which leads to huge number of MR rounds.

We propose a more relaxed algorithm, PR2_{MR} which does away with the height. The excess flow will be guided solely based on the distance heuristic label of the neighboring vertices. That is, the excess flows are pushed towards vertices that are closer to t. PR2_{MR} is resilient against the bad situation as depicted in Figure 4.3. Moreover, PR2_{MR} is robust and has overall significantly less number of MR rounds required compared to that of PR_{MR}.

The original Push-Relabel algorithm terminates when all the excess flows either reach t or flown back to s. Our modified PR2_{MR} terminates when no more excess flow can be pushed towards t (i.e., stuck) and no update can be performed for the distance labelling heuristics due to no more residual path connecting s to

Cluster	Nodes	CPU	Memory	Hard Disk
A	5	Xeon E5540 @ 2.83GHz x 8	16GB	73GB SAS
B	7	Opteron @ 2.2GHz x 4	2GB	73GB SCSI

Table 4.2: Cluster Specifications

t. Both of these conditions can be detected via event counters. To relate to the original Push-Relabel algorithm, the *stuck* excess flows will eventually flow back to s and the algorithm terminates. Our PR2$_{MR}$ does not require the stuck excess flows to be returned to s thus saving a number of MR rounds.

4.3.6 Experiment Results on PR$_{MR}$

We used Hadoop (0.20.1). Experiments were run using a heterogeneous cluster consisting of two kinds of nodes, A and B shown in Table 4.2. In this section, we evaluate the robustness and scalability of both our PR$_{MR}$ and PR2$_{MR}$ algorithms.

PR2$_{MR}$ Robustness

As mentioned in Section 4.3.4, PR$_{MR}$ algorithm can get into a bad situation when the excess flows are trapped because of push in wrong directions which leads to useless work and more MR rounds. We were only able to run PR$_{MR}$ on the FB0 graph. We selected 6 pairs of randomly selected source s and sink t vertexes from the FB0 graph.

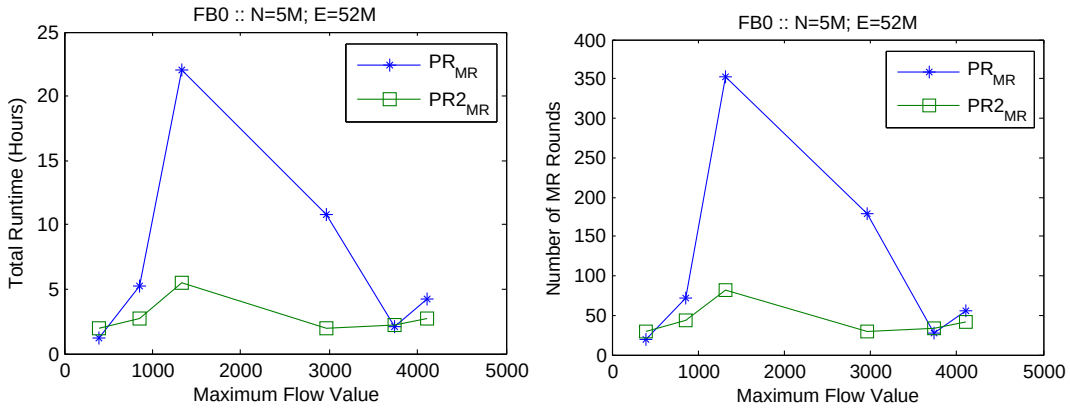


Figure 4.4: Robustness comparison of PR$_{MR}$ versus PR2$_{MR}$

Figure 4.4 shows the runtime and the number of MR rounds needed to compute the max-flow value for each of the 6 max-flow runs with different s and t pairs. We plot the 6 runs by increasing max-flow value on the x-axis. This experiment shows that the performance PR$_{MR}$ can be poor. One of 6 runs of PR$_{MR}$ exceeds the cut-off runtime of 24 hours (the PR$_{MR}$ run for the max-flow

value of 1328 is still not complete within 24 hours). On the other hand, PR2$_{MR}$, shows better runtimes by consistently finishing in less than 6 hours and less than 80 MR rounds for all the 6 runs.

PR2$_{MR}$ Flow Scalability

This experiment tests the effect of increasing the max-flow value on runtime and the number of MR rounds using the FB1 graph. In our graphs, the edge capacity is one for all edges and each vertex in the Facebook graph has at most 5000 edges. This gives an upper bound on the maximum-flow value from s to t which cannot be larger than the minimum degree of s and t.

We modified the graph to have a larger maximum flow. To create much bigger max-flow values, we select w vertices and connect them to a super source s. Similarly, we select another set of w vertices and connect them to a super sink t. The edge capacity from s and t to their connected vertices is set to infinity. To measure the effect of the max-flow value on runtime and the number of required MR rounds, we created several tests varying w from 1, 2, 4, 8, 16, 32, 64, and 128 vertices. The more the number of vertices w that are connected to s and t, the larger the max-flow value from s to t.

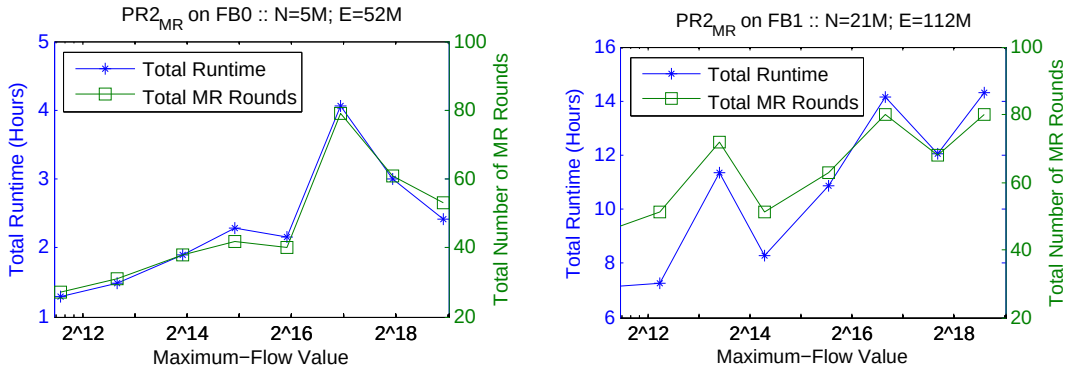


Figure 4.5: The Effect of Increasing the Maximum Flow and Graph Size

Figure 4.5 plots the runtime and MR rounds against the max-flow values on a logarithmic scale (x-axis). It shows the effects on the runtime and MR round as the graph size increases. The graph size of FB0 (left graph) and FB1 (right graph) is given in the Facebook subgraph table 4.1. We can see the scalability of the PR2$_{MR}$ as the graph increases is quite good. FB1 has 6 times the number of vertices and 4 times the number of edges, however the number of MR rounds required increases less than twice. The runtime increase proportionally as the graph size (roughly 4 times increase). We remark that we are using rather large flow values with very large graphs. If we had not used the special construction which gives arbitrary large flows, the runtimes would have been smaller given the

maximum degree limit of 5000 in Facebook. The number of required MR rounds does not seem to increase as fast as the graph gets larger. This may suggest that the larger the graph, the smaller the diameter which is also consistent with the findings in [67]. It also gives evidence that choosing to minimize the number of rounds is well suited for the kinds of real world graphs which come from social networks.

MR Resource usage on the FB0 and FB1

We measured the resources used while running the PR2_{MR} on the FB0 and FB1 graphs. The measurements are extracted from the Hadoop output terminal while the MR job is running. The output consists of values of all counters in the MR job. Some counters are defined by the user and the others are built in counters from Hadoop. The user counters are incremented during the execution of the user defined MAP function and REDUCE function while the Hadoop counters are incremented by Hadoop internally during the run. These counters can be read at the end of the MR job by the master node and can be used for termination conditions or monitoring.

We display the output of each MR round into a table to show the resources and the progress of each round. Among all the counters we are interested in the number of map input bytes (**mrmib**), map output bytes (**mrmob**), reduce shuffle bytes (**mrrsb**), our own counter $SINK\_EXCESS$ (**ex**), and the time to complete the round (**time**). Table 4.3 gives the counter values for each MR rounds when calculating the maximum-flow for a pair of vertices in the FB0 graph.

For our smaller graph, FB0, the original input size graph is about 157 MB which consists of 5 million vertices and 52 million edges. We can see this in the first row (r = 0) on *mrmib* column. This shows that the MAP function read input of 157 MB. However, the output of the MAP function exploded to 1.6 GB which can be seen on the *mrmob* column. This is expected since the original graph, FB0, contains only a pure list of uni-directed edges and the output of MAP function will annotate each vertex with additional attributes such as excess, timestamp, distance, and each edges will be turned into bi-directed edges with additional attributes such as id, flow, capacity, timestamp, and distance. Such additional data structures are required to run the PR_{MR} and the PR2_{MR} algorithm.

The *mrrsb* shows the number of bytes that are being shuffled across the compute nodes. The shuffled records are the intermediate key-value pairs that are produced by the MAP function. The records are compressed using gzip compression to lower the network traffic.

We can see the progress being made each round by observing the amount of

r	mrmib	mrmob	mrrsb	ex	time
0	156,699,849	1,638,202,592	381,809,770		00:02:26
1	1,009,546,692	962,862,136	307,793,522		00:01:32
2	1,009,546,692	963,191,056	309,551,388		00:01:33
3	1,009,546,692	969,213,416	312,022,025		00:01:37
4	1,009,546,692	1,540,395,616	419,774,494	10	00:02:08
5	1,009,546,692	2,271,389,656	574,022,914	17	00:02:40
6	1,009,546,692	1,149,689,576	360,101,336	738	00:01:45
7	1,009,546,692	984,980,056	330,466,610	835	00:01:37
8	1,009,546,692	1,239,468,416	378,130,641	1,274	00:01:48
9	1,009,546,692	2,627,067,696	645,034,572	1,381	00:02:51
10	1,009,546,692	2,892,539,576	710,913,982	1,694	00:03:08
11	1,009,546,692	2,958,724,336	700,551,374	1,745	00:03:18
12	1,009,546,692	2,914,803,976	690,154,558	2,126	00:03:24
13	1,009,546,692	2,960,309,376	692,551,589	2,246	00:03:13
14	1,009,546,692	2,830,300,256	689,195,862	2,453	00:03:10
15	1,009,546,692	2,873,117,496	697,233,739	2,586	00:03:03
16	1,009,546,692	2,810,039,976	690,792,372	2,688	00:03:05
17	1,009,546,692	2,725,248,456	673,217,150	2,822	00:03:02
18	1,009,546,692	2,853,901,096	704,065,116	2,895	00:03:03
19	1,009,546,692	2,789,270,576	673,073,968	2,955	00:03:04
20	1,009,546,692	2,823,891,096	698,017,310	2,979	00:03:18
21	1,009,546,692	2,871,790,256	688,370,051	2,995	00:03:45
22	1,009,546,692	2,839,478,736	687,244,651	3,011	00:03:40
23	1,009,546,692	2,779,275,736	679,825,607	3,023	00:03:03
24	1,009,546,692	2,565,547,496	627,649,875	3,036	00:02:56
25	1,009,546,692	2,643,215,136	646,138,567	3,042	00:03:05
26	1,009,546,692	2,641,973,656	644,207,946	3,043	00:02:57
27	1,009,546,692	1,943,816,216	527,132,944	3,043	00:02:31

Table 4.3: FB0 with $|f^*| = 3043$, Total Runtime = 1 hour 17 mins.

excess flow that arrived at t in ex event counter. The sink excess counter will be incremented whenever there is an excess that reaches the sink vertex. The first three rounds, no excess arrived to the sink vertex. In the fourth round there are 10 excesses reaches the sink vertex and the next round another 7 excesses arrives and so on. When the timestamp (along with the distance labels) has been propagated to all vertices and there is no more excess flow movements, the PR2_{MR} algorithm terminates and the latest value of the ex counter (the sink excess flow amount at t) is the maximum-flow value.

Table 4.4 shows the counter values for the FB1 graph. The FB1 graph initial size is 300 MB with 30 million vertices and 214 million edges. When the graph is annotated with the PR2_{MR} data structure, it grows as large as 3 to 6 GB. We can see that the time per round increased linearly with the graph. FB1 has 4 times the number of edges compared to FB0 and the runtime for FB1 roughly 4 times slower than FB0.

r	mrmib	mrmob	mrrsb	ex	time
0	300,602,700	3,224,346,080	757,961,865		00:05:39
1	2,481,989,368	2,296,346,456	687,160,761		00:06:02
2	2,481,989,368	2,296,683,296	687,233,843		00:07:09
3	2,481,989,368	2,300,192,456	695,928,889		00:07:15
4	2,481,989,368	2,385,868,776	709,923,129		00:06:35
5	2,481,989,368	2,823,052,936	800,393,751		00:06:50
6	2,481,989,368	4,647,440,136	1,209,425,235	8	00:10:38
7	2,481,989,368	3,708,317,256	1,023,563,604	8	00:09:56
8	2,481,989,368	2,386,338,216	753,020,686	36	00:09:12
9	2,481,989,368	2,298,626,296	728,122,607	85	00:05:07
10	2,481,989,368	2,368,599,176	741,197,114	204	00:07:33
11	2,481,989,368	2,604,512,896	794,587,588	248	00:08:06
12	2,481,989,368	4,189,866,336	1,142,621,127	335	00:09:23
13	2,481,989,368	4,651,530,976	1,222,046,728	354	00:09:14
14	2,481,989,368	4,521,581,816	1,219,731,150	427	00:11:00
15	2,481,989,368	6,407,965,056	1,599,999,331	457	00:12:03
16	2,481,989,368	6,664,348,256	1,648,231,639	567	00:16:02
17	2,481,989,368	6,554,123,656	1,625,028,832	605	00:12:04
18	2,481,989,368	6,624,325,216	1,640,516,363	664	00:14:10
19	2,481,989,368	6,685,504,416	1,645,943,916	717	00:10:55
20	2,481,989,368	6,620,344,456	1,633,835,506	779	00:12:12
21	2,481,989,368	6,663,377,696	1,650,748,412	808	00:11:37
22	2,481,989,368	6,717,187,056	1,659,466,508	854	00:14:11
23	2,481,989,368	6,739,214,776	1,659,520,665	866	00:11:05
24	2,481,989,368	6,739,059,376	1,656,906,839	876	00:13:17
25	2,481,989,368	6,743,986,016	1,664,217,994	884	00:23:12
26	2,481,989,368	6,752,191,616	1,658,298,808	889	00:12:39
27	2,481,989,368	6,758,444,216	1,667,007,616	890	00:12:54

Table 4.4: FB1 with $|f^*| = 890$, Total Runtime = 6 hours 54 mins.

4.3.7 Problems with PR_{MR} and $PR2_{MR}$

Although the Push-Relabel algorithm would appear to fit into a distributed parallel computing setting, we found it unsuitable for MR for two main reasons. First, Push-Relabel appears to have low available parallelism when there are only a few *active elements* (i.e., nodes with positive excess flow) left in the graph [77], which can be common. In the MR setting, this means that if we have thousands of mappers and reducers running in parallel, only a small fraction of them might be doing useful work (i.e., only a few workers contribute in excess flow transfer). The remaining workers will perform unproductive work (i.e., serializing/deserializing the records without any contribution to the completion of the task). Second, the Push-Relabel algorithm relies heavily on heuristics to decide which vertices to push the excess flow to [23]. A wrong push can lead to a long chain of excess flow transfer (i.e., the excess flow can wander around in a dead-end subgraph). In an MR setting, a vertex can only *push* information in one round. It cannot request or *pull* the state of other vertices. Thus, excess flow transfer from one

vertex to another would take one MR round which is expensive in MR. A long chain of excess flow transfer directly increase the number of required MR rounds. While we were able to minimize the number of rounds by relaxing the height constraint as in PR2$_{MR}$ algorithm, it still not practical to be used in processing much larger graph sizes. To scale to far larger graph sizes, we designed our own MapReduce-based max-flow algorithms based on the Ford-Fulkerson method.

In the next section, we detail our designs, implementations, optimizations and evaluations of our MR-based max-flow algorithms based on the Ford-Fulkerson method.

4.4 A MapReduce-based Ford-Fulkerson Method

As briefly mentioned in Section 4.2.3, the *sequential* Ford-Fulkerson method works by successively finding an augmenting path from vertex s to vertex t. Each time an augmenting path is found, the flow along the path is *augmented*. When an augmenting path is augmented, the flow of the edges along the path will be increased by the minimum residual capacity of the edges along the path and the reverse flow of the edges will be decreased by the same amount. This will change the flow in the residual network, it can possibly cause some edges to be saturated and removed from the residual network (and conversely, some edges can become un-saturated and added back to the residual network). This is repeated until no more augmenting paths can be found. The pseudocode is given in Figure 4.6 and an example of finding the max-flow using the Ford-Fulkerson method is illustrated in Figure 4.7.

while true **do**
  P = find an augmenting path in G_f
  if (P does not exist) **break**
  Augment the flow f along the path P

Figure 4.6: The Ford-Fulkerson method

One way to find an augmenting path in the current residual network is to run a Breadth-First Search (BFS) traversal from s, visiting vertices level by level. In the first level, the neighbors of the source s are visited. In the second level, the neighbors of the neighbors of s are visited and so on. The search keeps track of the path from s to the visited vertex, so that when vertex t is visited, an augmenting path is found. In a MR setting, each level of BFS can be done in a single round. If the residual network has diameter D, then a MR-based BFS from s takes $O(D)$ rounds to complete.

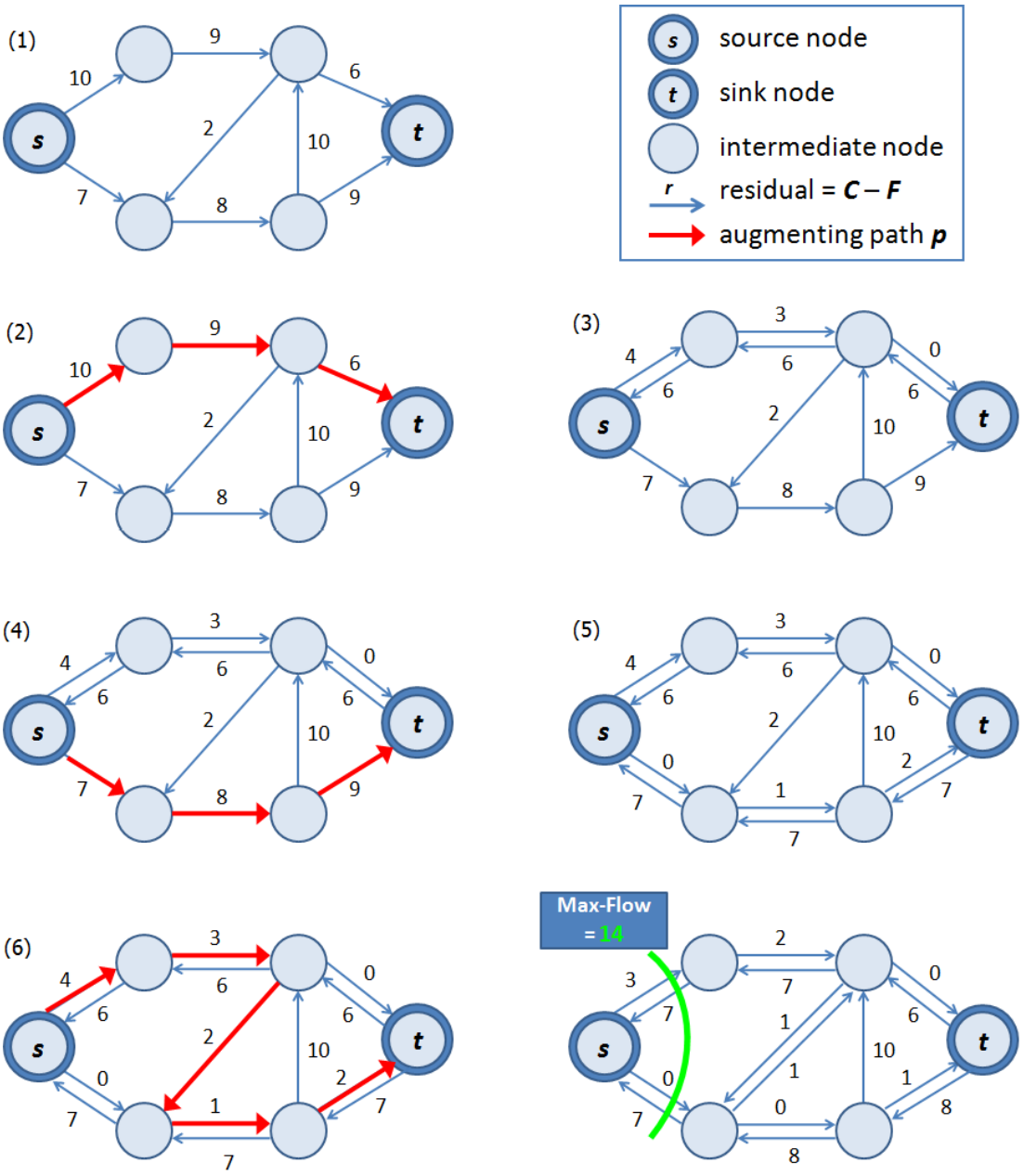


Figure 4.7: An Illustration of the Ford-Fulkerson Method

Figure 4.7 shows an example of finding the max-flow using the Ford-Fulkerson method. At the top left corner is the initial residual network (1). The edges in the graph represent the residual edges and the weight of the edge is the residual capacity of the edge. Initially, all the edges have zero flow, therefore the residual capacity of all the edges are equal to its capacity. The residual network labeled (2) shows an augmenting path P is found and (3) shows the updated residual network after P has been augmented with flow amount $\delta = 6$. The next augmenting path is found on the updated residual network (4) and augmented (5) with $\delta = 7$. Finally, the last augmenting path is found (6) and augmented with $\delta = 1$. The

Ford-Fulkerson method terminates with the max-flow value $|f^*| = 14$.

The kinds of real-world graphs that we consider have small-world properties (i.e., small expected diameter D) which allows us to effectively use MR graph algorithms based on BFS. Nevertheless, a direct conversion of Ford-Fulkerson method to MR can lead to $O(|f^*|D)$ rounds since for each flow increase, $O(D)$ rounds[1] might be needed. This is not practical for large max-flow values. Consider a real small-world graph with 1 billion edges, running Hadoop on 5 machines in our cluster requires at least 10 minutes to complete one round. Assuming $D \approx 10$ and a max-flow value of ∼400K, it would require about ∼4M rounds and ∼75 years to finish on our cluster. In this section, we show how we can parallelize the Ford-Fulkerson method by incrementally finding augmenting paths and then further increase parallelism with bi-directional search and multiple excess paths. We show how to extract large amounts of parallelism, so that we are able to find many augmenting paths in a single round and also in subsequent rounds. This reduces the number of required rounds tremendously. In the 1 billion edge graph example, our MR algorithm requires only 9 rounds and ∼2 hours (see Figure 4.20).

We call the initial design of our max-flow MR algorithm given in this section, FF1. Later in Sec. 4.5, we optimize further to get other variants, FF2 to FF5. We emphasize that we are interested in average rather than the worst case complexity as we want to obtain practical max-flow implementations on actual real-world graph instances, i.e., the growing graph of a social network such as Facebook.

4.4.1 Overview of the FF$_{MR}$ algorithm: FF1

```
1.  round = 0
2.  while true do
3.     job = new Job() // create a new MapReduce job
4.     set the job's MAP and REDUCE class, input
5.        and output path, the number of reducers, etc.
6.     job.waitForCompletion() // submit the job and wait
7.     c = job.getCounters() // event counters
8.     som = c.getValue("source_move");
9.     sim = c.getValue("sink_move");
10.    if (round > 0 ∧ (som = 0 ∨ sim = 0)) break
11.    round = round + 1
```

Figure 4.8: The pseudocode of the main program of FF1

[1] D is not constant as the residual network may change in each round. We assume the underlying graph is robust and dense enough that the diameter stays small. This properties will be evaluated in our experiments Section 4.7.

We start with an overview of the main program of the FF1 algorithm in Fig. 4.8 which performs a number of rounds of MR jobs. For each round, a new MR job is created. The job is assigned a MAP and a REDUCE function, input/output path in the DFS_{MR}, etc. (line 4-5). After the job is configured, it is sent to the master node to be scheduled for execution (line 6) and the main program blocks until the job is finished. During the job, custom counters are created and incremented inside MAP or REDUCE which are available to the main program after the job is finished (line 7). Counters are used as sentinels, or for changing the strategy in the next round.

We use the first round of MR to convert the input graph into our graph data structure (see Sec. 4.4.3), make the edges bi-directional and initialize the flow and capacity of each edge. Round #1 onwards use the MAP and REDUCE function given in Sec. 4.4.4 and Sec. 4.4.5. After round #0, the input for the current MR round is taken from the output of the previous round.

4.4.2 FF1: Parallelizing the Ford-Fulkerson Method

Several issues need to be addressed in designing an effective multi-round MR max-flow algorithm. First, as performing one round is expensive, we want to reduce the number of rounds (i.e., find and augment as many augmenting paths as possible in a round and in subsequent rounds). Second, recall that in MR model, each vertex can only push information to another vertex in one round. Each vertex will receive the information pushed from other vertices in the next round. Hence, we want to use the information in the current round effectively, rather than deferring its computation to the next round. Third, for jobs with large data sizes (e.g. MR jobs), it is common that the compute time is less than the time to fetch the data. We define a vertex to be *active* if it has something to compute. In executing an MR job, all vertices will be read, shuffled, and written back to disk regardless of whether they are active or not. Thus, a vertex should be active to get useful computation from the MAP and REDUCE. Ideally, we want the MR algorithm to scale linearly as we add more machines. However, this would only happen if the algorithm has sufficiently high available parallelism, i.e., we want the number of active vertices (or active elements [77]) to be large compared to the available computing resources (number of mappers/reducers).

We solve the parallelism problem by using speculative execution. The idea is to organize for a MAP on a vertex to try to do some work. In this case, to always try to extend a path. This is speculative since some of the work (i.e., finding excess or augmenting paths) may be discarded at a later time. The significance of the speculation is that it both increases parallelism by making more vertices

active and shifts work which may otherwise happen in later rounds to earlier rounds.

FF1, our initial design, speculatively finds augmenting paths concurrently which can significantly reduce the number of rounds required to compute max-flow, it then increases the available parallelism with bi-directional search as well as maintaining the high degree parallelism with multiple excess paths.

FF1 (a) : Finding Augmenting Paths Incrementally

To find an augmenting path, we need to find a path from s to t in the residual network. We define an *excess path* of a vertex as a path from s to that vertex. Initially, vertex s is the only vertex which has an excess path. Each round, each vertex that has an excess path will extend it to its neighbor (avoiding cycles). By definition, a vertex that has an excess path is an *active* vertex because it has something to compute (i.e., to push information to its neighbors). When an excess path is extended to t, an augmenting path is found. In a round, it is possible that several neighbors of t send their excess path to t, thus t may receive more than one augmenting path. The reducer processing vertex t decides (locally) whether to accept/reject these augmenting paths. We want to accept as many augmenting paths as possible in one round to avoid spilling the work to the next round. At the end of the round, all augmenting paths that are accepted by t are augmented.

Augmenting paths that are augmented in the current round change the flow of some edges. For each vertex to have a consistent view of the residual network, the flow changes must be broadcast to every vertex in the next round. This can be achieved by distributing a list of the augmented edges and its Δ flow (generated by the reducer processing t in the current round) to all the mappers in the next round. The size of the list is proportional to the flow changes and is expected to be much smaller than size of the graph. We implement the list as an external file, rather than as MR output as it can be viewed as global data generated from the current round which all mappers read in the next round. The mappers in the next round apply the flow changes in parallel to each affected edge in the residual network. The flow changes may cause some excess paths to be saturated (i.e., some edges in the path becomes saturated). Non-saturated excess paths on a vertex can continue to be extended to its neighbors while saturated excess paths are removed from the vertex.

The above technique incrementally updates the residual network in subsequent rounds by reusing the computation of the previous round rather than starting anew, thus maintaining high parallelism in MR (i.e., keeping the number of active vertices high in subsequent rounds) allows augmenting paths to be found more

rapidly in the subsequent rounds. We expect the incremental update to find augmenting paths continuously in the subsequent rounds thus may lower the number of rounds from $O(|f^*|D)$ down to $O(|f^*|)$ rounds.

The incremental finding of augmenting paths speculatively extends excess paths of each vertex to all of its neighbors. This rapidly increases the amount of active vertices in subsequent rounds which contributes to the highly parallel nature of the algorithm. Moreover, it is also used to decide on the termination of the algorithm when no more excess paths can be extended (see Section 4.4.6).

FF1 (b) : Bi-directional Search

In the first few rounds, only the source vertex s and its neighbors are active (few active vertices compared to the total number of vertices). To increase parallelism, we introduce an analogous excess path from the sink vertex t. We define a *source excess path* of a vertex as a path from source s to the vertex in the residual network. Similarly, a *sink excess path* of a vertex is defined as a path starting from the vertex to sink t in the residual network. In the first round, s starts extending its source excess path, while t starts extending its sink excess path. Bi-directional search helps in doubling the amount of available parallelism as the use of the sink excess doubles the number of active vertices, at least for the first few rounds. Moreover, it may halve the total number of rounds as we do not need to wait until a source excess path reaches t to find an augmenting path.

Using bi-directional search, any vertex u may have both a source and sink excess path, thus can generate an augmenting path. This allows a huge number of augmenting paths generated in a round. Combined with the incremental updates strategy (FF1 (a)), the expected complexity of using the bi-directional search is $O(|f^*|/A)$ rounds, where A is the average number of augmenting paths accepted per round. With this, the bi-directional search becomes the most important search strategy towards making the FF1 practical as it effectively lower the number of rounds required down to $O(D)$ (see Section 4.7.1).

FF1 (c) : Multiple Excess Paths

Whenever an augmenting path is accepted in the previous round, some (source / sink) excess paths belonging to some vertices in the current round are dropped as some of the edges in the excess path is saturated. Thus, some vertices could lose excess paths making them inactive for the current round as they wait for an excess path from their neighbours (if any). To avoid such loss of parallelism (i.e., prevent a vertex from losing all of its source/sink excess paths), we allow each vertex to store multiple source and sink excess paths. We avoid space explosion

by limiting the maximum number of excess paths stored in each vertex to k and employ an accumulator (see Sec. 4.4.3) to decide locally which excess paths are stored. The larger the k, the less likely a vertex will become inactive when the residual network changes, however, the overhead for reading, writing, updating the excess paths also increases. We decided to only pick one of the k excess paths to be extended as experiments show that extending more than one excess path incurs overhead without much benefit. Multiple excess paths amplify the effectiveness of the previous strategies (FF1 (a) and FF1 (b)) by making vertices active for a longer time and contribute towards lowering the number of rounds further (see Section 4.7.1).

4.4.3 Data Structures for FF$_{MR}$

We model the flow network as $\langle key, value \rangle$ records in the DFS$_{MR}$ where records represent vertex data structures. The *key* is the vertex ID (identifier) of a vertex u and the *value* is the tuple $\langle S_u, T_u, E_u \rangle$ consisting of: a list of source excess paths S_u; a list of sink excess paths T_u; and a list of edges E_u connecting u to its neighbors where each edge in E_u is a tuple $\langle e_v, e_{id}, e_f, e_c \rangle$ consisting of the vertex ID of the neighbor connected through the edge e_v, the edge ID e_{id}[2], the flow amount e_f from u to e_v, and the capacity of the edge e_c. The source excess paths S_u is a list of excess paths from s to u. Similarly, sink excess paths T_u is a list of excess paths from u to t. Each excess path in S_u and T_u is a list of edges containing a sequence of edge IDs along with the flow and capacity of the edges along the sequence.

The advantage of modelling a record as a vertex is that it is in line with the recently proposed graph processing framework based on the bulk synchronous parallel model [82] which is also a vertex-centric processing. The disadvantage of modelling a record as a vertex is that there can be huge variability in the degree of the vertices. In small-world graphs, there can be vertices with arbitrarily large degrees which cause huge variations in the workloads for the workers. However, as we mentioned in Section 4.2.4, vertices with degree larger than C can be decomposed into a number of vertices such that every vertices have degree at most C without affecting the max-flow value.

Another way of modelling is to store each edge as a record (i.e., edge-list data structure). While this has the advantage that each edge record will have roughly identical size, it is difficult or even impossible to process such structure

[2] We require an efficient way to identify each edge by ID for fast lookup to determine whether an excess path contains a certain edge. We give each edge (u,v) an unique ID that is the concatenation of the vertex IDs of both endpoints in lexicographic order. Determining whether an edge exists in an excess path can be done in $O(1)$ using a hash-table.

because of the lack of useful information. That is, how does an edge extends its excess path/flow to its neighbors without having a list of neighbors? If the list of neighbors is included for every edge, then it will have the same disadvantage of the degree variability problem as in the vertex-based model. Thus, we argue that vertex-based data structure is the most suitable model.

It has been shown that a poor choice of augmenting paths "can lead to severe computational cost" [28]. An optimization such as selecting the shortest augmenting paths can give a strongly polynomial algorithm $O(VE^2)$. Moreover, by building a layered network to quickly find the flows of all shortest augmenting paths (blocking flow), a better complexity $O(V^2E)$ [26] can be achieved.[3] Unfortunately, these optimizations require a global view of the graph and are not compatible with the MR model which requires a local view. Nevertheless, our strategies presented in Sec. 4.4.2 are related to ideas in [28, 26] as shorter/earlier augmenting paths found will be augmented before the longer ones.

A number of candidate augmenting paths can be found in one MR round, but it might not be possible for all of them to be augmented due to conflicting augmenting paths. Two augmenting paths conflict if there is a common edge shared by the two augmenting paths such that if both augmenting paths are augmented, the flow of the edge will violate the capacity constraint (i.e., the edge flow becomes larger than its capacity). In this case, only one of the conflicting augmenting paths can be augmented. The other augmenting paths have to be rejected. For a similar reason, storing multiple conflicting excess paths in a vertex is ineffective, hence, the excess paths in a vertex should be conflict-free.

The Accumulator

The decision to reject an excess/augmenting paths can be made locally (i.e., it does not require the global state of the graph) since all vertices have the same and consistent view of the current residual graph. We introduce an *accumulator* data structure for this task. It greedily "accepts" non-conflicting excess paths on a first-come-first-serve basis. Initially, the accumulator is empty and it is later filled in by excess paths that are accepted. To test whether an excess path can be accepted, the accumulator uses the currently accepted excess paths and checks for capacity constraint violation. If no violation is detected[4], the excess path will

[3] [36] gets a complexity of $O(min(V^{2/3}, E^{1/2})E \log(V^2/E) \log U)$ where U is the maximum edge capacity with residual flow upper bounds.

[4] For example, consider an augmenting path $s-a-b-c-d-t$ and another augmenting path $s-u-b-c-v-t$. Each edge along the augmenting paths has residual capacity of one, so only one of the augmented paths can be accepted, otherwise edge $b-c$ will be used twice which violates the capacity constraint. However, there is no problem with accepting another augmenting path $s-c-b-t$ after one of the above augmenting paths is accepted since $b-c$ and $c-b$ are in opposite directions. They cancel the flow of the edge, thus, do not violate the

be accepted and stored in the accumulator. The same accumulator can be used to decide the acceptance of candidate augmenting paths since an augmenting path is just a special excess path from s to t. We remark that it does not make sense to have an "ideal accumulator" that accepts as many excess/augmenting paths as possible since each vertex can only have a local view of the graph. To ensure all possible excess/augmenting paths are explored, each vertex will have to keep generating excess/augmenting paths in subsequent rounds.

4.4.4 The MAP Function in the FF1 Algorithm

The MAP function for FF1 is given in Fig. 4.9. Its job is (a) to update the current residual network based on the previous round's flow changes, (b) to generate new augmenting path *candidates* based on the updated residual graph as well as (c) extending excess paths in each vertex to its neighbors. We use tuple notation ($\langle\rangle$ denotes the empty set) to represent sets since the sets are stored as tuples in MR records. The notation $a|b$ means concatenate path a with path b. The MAP function takes a record (representing a vertex) with $key = u$, $value = \langle S_u, T_u, E_u\rangle$ and performs three operations:

function MAP$_{FF1}(u, \langle S_u, T_u, E_u\rangle)$
1. **foreach** $(e \in S_u, T_u, E_u)$ **do** // *update all edges*
2. a = AugmentedEdges[round-1].get(e_{id})
3. **if** (a exists) $e_f = e_f + a_f$ // *update edge flow*
4. Remove saturated excess paths in S_u and T_u
5. A = new Accumulator() // *local filter*
6. **foreach** ($se \in S_u$, $te \in T_u$) **do**
7. **if** ($A.accept(se|te)$) // *se|te is an augmenting path*
8. EMIT-INTERMEDIATE$(t, \langle\langle se|te\rangle, \langle\rangle, \langle\rangle\rangle)$
9. **if** ($S_u \neq \langle\rangle$) // *extend source excess path if it exists*
10. **foreach** ($e \in E_u$, $e_f < e_c$) **do**
11. se = pick one source excess path from S_u
12. EMIT-INTERMEDIATE$(e_v, \langle\langle se|e\rangle, \langle\rangle, \langle\rangle\rangle)$
13. **if** ($T_u \neq \langle\rangle$) // *extend sink excess path if it exists*
14. **foreach** ($e \in E_u$, $-e_f < e_c$) **do**
15. te = pick one sink excess path from T_u
16. EMIT-INTERMEDIATE$(e_v, \langle\langle\rangle, \langle e|te\rangle, \langle\rangle\rangle)$
17 EMIT-INTERMEDIATE$(u, \langle S_u, T_u, E_u\rangle)$

Figure 4.9: The MAP function in the FF1 algorithm

Update All Edge Flows (MAP$_{FF1}$:1-4). A vertex has a collection of edges in S_u, T_u, and E_u. All the edge flows are updated according to the Δ flow

capacity constraint.

changes collected from the previous round's augmented edges using a AugmentedEdges hash-table (read-only in mappers) to lookup its edge ID, e_{id} (MAP$_{FF1}$:2). If AugmentedEdges returns an edge a (containing the Δ flow of edge a), edge e will be augmented using a's flow (MAP$_{FF1}$:3). After all edges in the vertex have been updated, some excess paths in S_u or T_u may be saturated and are removed (MAP$_{FF1}$:4).

Generate Augmenting Paths (MAP$_{FF1}$:5-8). If a vertex has at least one source and sink excess path, then concatenating them together gives an augmenting path (MAP$_{FF1}$:6-7). We use an accumulator to locally reject augmenting paths whose acceptance will violate the capacity constraint. The accepted augmenting paths are sent to t (MAP$_{FF1}$:8) as candidate paths which may be rejected further in the reduce phase if they conflict with other augmenting paths from other mappers.

Extending Excess Paths (MAP$_{FF1}$:9-16). If a vertex has source excess paths, some of them will be extended to all neighboring vertices (MAP$_{FF1}$:9-12). We pick a source excess path and ensure no cycle is formed if it is to be extended with edge e (MAP$_{FF1}$:11). The neighboring vertex (e_v) is notified of the extended source excess path (MAP$_{FF1}$:12). Sink excess paths are extended similarly (MAP$_{FF1}$:13-16).

Each vertex is represented as a record. We call the record representing a vertex, i.e., containing the vertex edges, source and sink excess paths, the *master* vertex record (or simply, the master vertex). When the MAP function processes a master vertex, it emits intermediate records which we call *vertex fragments* (or simply, fragments) (MAP$_{FF1}$:8,12,16) which are designated to other vertices. Vertex fragments do not contain edge information. The master vertex itself is also emitted (MAP$_{FF1}$:17). Both master and fragment records will be output as intermediate records. We can think of the map phase as a way to push information from one vertex to other vertices by emitting vertex fragments.

4.4.5 The REDUCE Function in the FF1 Algorithm

The REDUCE function in FF1 given in Fig. 4.10 processes all fragments of each vertex with its master emitted during the map phase. If the current vertex being processed is the sink t, then all the augmenting paths candidates generated during the map phase will be re-checked for conflicts then the accepted augmenting paths are augmented. In addition, a file (the AugmentedEdges hash-table) containing the flow changes in the current round is generated to be used by mappers in the next round to update the residual graph.

REDUCE aggregates all intermediate records having the same key that are output by MAP$_{FF1}$. The aggregation iterates through the list of values $\langle S_v, T_v, E_v \rangle$

function REDUCE$_{FF1}(u, values)$
1. A_p, A_s, A_t = new Accumulator()
2. $S_m = T_m = S_u = T_u = E_u = \langle\rangle$
3. **foreach** $(\langle S_v, T_v, E_v\rangle \in values)$ **do**
4. **if** $(E_v \neq \langle\rangle)$ $S_m = S_v$, $T_m = T_v$, $E_u = E_v$
5. **foreach** $(se \in S_v)$ **do** // *merge / filter* S_v
6. **if** $(u = t)$ A_p.accept(se) // *se = augmenting path*
7. **else if** $(|S_u| < \mathbf{k} \wedge A_s.accept(se))$ $S_u = S_u \cup se$
8. **foreach** $(te \in T_v)$ **do** // *merge / filter* T_v
9. **if** $(|T_u| < \mathbf{k} \wedge A_t.accept(te))$ $T_u = T_u \cup te$
10. **if** $(|S_m| = 0 \wedge |S_u| > 0)$ INCR('source_move')
11. **if** $(|T_m| = 0 \wedge |T_u| > 0)$ INCR('sink_move')
12. **if** $(u = t)$ // *collect all augmented edges in* A_p
13. **foreach** $(e \in A_p)$ **do**
14. AugmentedEdges[round].put(e_{id}, e_f)
15. EMIT$(u, \langle S_u, T_u, E_u\rangle)$

Figure 4.10: The REDUCE function in the FF1 algorithm

containing the (master) vertex and its fragments emitted by other vertices during the map phase. The master vertex is differentiated from a vertex fragment as it has at least one edge (REDUCE$_{FF1}$:4). When sink t is reduced, all excess $path \in S_v$ are augmenting path candidates. Conflicting augmenting path candidates get filtered by an accumulator A_p (REDUCE$_{FF1}$:6). For vertices other than t, an accumulator A_s is used to locally reject and store at most k non-conflicting source excess paths (REDUCE$_{FF1}$:7). Merging the sink excess paths is similar to merging source excess paths using an accumulator A_t (REDUCE$_{FF1}$:8-9).

We define *movement* of source excess path to denote when a vertex does not have a source excess path (or all its source excess paths are saturated due to residual graph changes) at the beginning of the round and gains at least one source excess path at the end of the round. Sink excess path *movement* is defined similarly. We employ two event counters to record movement, 'source_move' and 'sink_move' (REDUCE$_{FF1}$:10-11), which are used to determine when to terminate the algorithm (Fig. 4.8:10). The 'source_move' counter is incremented (using the MR operation INCR) when the master vertex u does not have any source excess path before merging is done (REDUCE$_{FF1}$:4) and acquires at least one source excess path after merging with its fragments (REDUCE$_{FF1}$:10) . The 'sink_move' counter is computed similarly.

The reducer processing the sink vertex t finalizes the acceptance of the augmenting paths in this round. The edges of accepted augmenting paths (stored in accumulator A_p) are added to the AugmentedEdges hash-table for this round (REDUCE$_{FF1}$:12-14). This hash-table associates the edge ID e_{id} with its Δ flow e_f

and is stored as a file in DFS_{MR} which is written when the reducer for sink t finishes. It is only read by the MAP_{FF1} function of the next round. Finally, the updated vertex u is emitted as the final output record for this round (REDUCE_{FF1}:15), which will be used as the input for the next round.

4.4.6 Termination and Correctness of FF1

The maximum flow is reached when no more augmenting paths can be found in the residual network [30]. While our algorithm works by finding augmenting paths locally from the vertex's local view, we also ensure that all possible augmenting paths are explored by tracking the local movements of the excess paths via the MR event counter in each round. FF1 terminates when either the *source_move* or *sink_move* counter is zero at the end of a round (see Fig. 4.8). In the former, it means that no source excess path can be extended and similarly, no sink excess paths can be extended for the latter. If neither a source nor sink excess path can be extended, it means that no more augmenting paths can be produced. Thus, the algorithm terminates with the maximum-flow.

In the MR reduce phase, records having the same intermediate key will go to the same reducer. The reducer that processes vertex t will be the only worker that decides which augmenting paths to be accepted. The decision to accept the augmenting paths is done sequentially, hence, there are no data races. However, this reducer becomes the bottleneck as the number of augmenting path candidates becomes very large. We address this bottleneck in the next section.

4.5 MapReduce Extension and Optimizations

Job execution in MapReduce involves overheads from the framework itself. The overhead from reading/writing to DFS_{MR} and the shuffle/sort between mappers and reducers is non-trivial and possibly larger than the computation in the mappers/reducers. In this section, starting with the baseline FF1 algorithm, we identify bottlenecks and design more optimizations to make MR more efficient and to increase parallelism. To summarize the MR optimizations from the baseline FF1: FF2 uses a stateful accumulator process outside MR which can be thought of as an "extension" for the MR framework; FF3 uses a variant of the schimmy design pattern [80]; FF4 eliminates object instantiations; and FF5 exploits the tradeoffs between storage versus number of rounds and number of intermediate records. We remark that the ideas of these optimizations may be applied for other graph algorithms for MR.

4.5.1 FF2: Stateful Extension for MR

The philosophy of MR is that the MAP and REDUCE functions should be designed to be stateless to allow easy distribution and scalability. In the Ford-Fulkerson method, stateful execution is needed when determining whether an augmenting path should be accepted. For simplicity in a distributed system, FF1 used a sequential and stateful augmenter for deciding augmenting paths acceptance.

The FF1 augmenter works by sending all augmenting paths found during the MAP phase to the sink vertex t. We use "send" informally to mean mappers and reducers interacting through (intermediate) records. During the REDUCE phase (REDUCE$_{FF1}$:6 in Fig. 4.10), only the reducer operating on vertex t will receive all augmenting paths found in that round. It then processes the acceptance sequentially using the accumulator data structure. As the number of augmenting paths increases, the reducer handling vertex t will become a processing bottleneck – completing much later than the other reducers. Moreover, the memory requirement for that reducer can be far higher than the rest of the reducers since arbitrarily large number of augmenting paths can arrive from any vertex in the residual network.

In the FF2 algorithm, we mitigate this problem by using an external process, called *aug_proc*, specially for accepting augmenting paths. Candidate augmenting paths can be generated in any vertex that has both source and sink excess path (MAP$_{FF1}$:6-8 in Fig. 4.9). Rather than generating it in the MAP function as in FF1, FF2 generates it in the previous round's REDUCE function. Each reducer establishes a persistent connection[5] to *aug_proc* and the REDUCE function sends augmenting paths found to *aug_proc* as soon as they are found. *aug_proc* receives augmenting paths and inserts them to a processing queue and returns immediately to avoid delaying the reducer. It has a thread that consumes augmenting paths from the processing queue to decide on acceptance using the accumulator. Thus, any vertex being processed by a reducer that has an augmenting path contacts the remote *aug_proc* directly rather than sending it via MR intermediate records through vertex t.

The advantages of using a dedicated process (*aug_proc*) outside MR are:

- Shrinks the size of the largest record. The record with $key = t$ can be extremely large as it contains all the augmenting path candidates (e.g. $> 10^5$ augmenting paths). With *aug_proc*, we can exclude storing the augmenting path candidates from vertex t's record, thus significantly shrink the size of the record t. Reducing the size of the biggest record lowers the memory requirements for the workers allowing more workers to run on a machine in

[5]Implemented using Java RMI

parallel.

- More augmenting paths can be send to the stateful accumulator. This is because we are no longer restricted to the excess paths limit k for vertex t (as in FF1). This may increase the number of augmenting paths accepted in a round, which in turn, increases the likelihood of needing fewer rounds to find the max-flow.

- Removes the FF1 bottleneck in accepting augmenting paths. Incoming augmenting path candidates are generated (uniformly) throughout the reduce phase across the workers and get processed soon after it is enqueued in *aug_proc*. Our experiments show that even with extremely large augmenting path candidates, the maximum queue size in *aug_proc* manages to stay small (see Table 4.5). *aug_proc* is not a bottleneck as it finishes immediately after the last reducer. *aug_proc* also eliminates the need to shuffle intermediate records (containing augmenting paths) to vertex t, which substantially reduces the MR shuffle-and-sort overheads. Any overhead from communicating with external resources (*aug_proc*) from the (isolated) REDUCE function is offset by the benefits.

4.5.2 FF3: Schimmy Design Pattern

We added an optimization step using the schimmy design pattern [80] which prevents the master vertices of the graph from being emitted as intermediate records during the map phase (see the MAP_{FF1} function in Fig. 4.9 line 17) thus reducing the MR-shuffle overhead. In the reduce phase, the reducers use the "schimmy" technique to merge the master vertices with the intermediate records. [80] also proposed to use in-mapper combining to reduce the MR-shuffle overhead but this not applicable in our case as the size of the intermediate records can far exceed the mapper's memory. Moreover, we do not use any combiners as we found worse performance.[6]

4.5.3 FF4: Eliminating Object Instantiations

In implementing MAP and REDUCE, it is important to avoid instantiating new objects with short lifetime as it burdens the garbage collection process. This is particularly relevant for Hadoop as the MR MAP and REDUCE functions are in Java. The FF4 algorithm achieves this by allocating the necessary data structures with fixed sizes and pre-allocating all objects. In MAP and REDUCE, the

[6] As a rule of thumb, combiners are only cost-effective if the map output can be aggregated sufficiently, i.e., by 20-30%. [8].

current record being processed replaces the content of the pre-allocated objects. Experiments in Sec. 4.7.4 show 1.1x - 1.4x runtime improvements depending on how many objects are created during in all rounds.

4.5.4 FF5: Preventing Redundant Messages

In FF1, we stored a limited number (k) of excess paths in a vertex to prevent space explosion, which causes the need for the excess paths to be re-sent in every round. Suppose a vertex V_1 wants to extend one of its excess path P_1 to its neighbor V_2. It is possible for P_1 to be rejected by V_2 simply because V_2 already accepted k excess paths from its other neighbors and therefore has no space left to store P_1. However, in the next round, some of the excess paths in V_2 may get saturated and V_2 will then have some space. Since V_1 doesn't know the status of V_2's storage (i.e., V_1 doesn't know whether P_1 was accepted or not), V_1 will have to re-send P_1 (or any other excess path) to V_2 at every round. This generates redundant messages increasing communication overhead in subsequent rounds.

There are two strategies we can employ to prevent the redundant messages. The first is to have V_2 notify V_1 whether it has accepted P_1, but this confirmation will cost additional one MR round and a communication message overhead. The second strategy is to set k to be the number of incoming edges of the vertex. This ensures that whenever a vertex extends one of its excess paths to its neighbor, there will be a space to accept. However, V_1 will have to remember which excess paths have been extended and to which neighbors to avoid re-send any other excess paths in the subsequent rounds. The cost is a small additional state flag for each excess path. V_1 will have to monitor all of its extended excess paths for saturation. If V_1 discovers that P_1 (which has been extended to V_2) has been saturated, then V_1 can pick another of its unsaturated excess paths (if any) and extend it to V_2.

In FF5, we employ the second strategy to prevent redundant messages as the overhead of the second strategy is far lower. This optimization significantly reduces the MR-shuffle overhead by preventing redundant intermediate records being shuffled across machines in subsequent rounds at the expense of a small increase in record sizes for the state information (see Sec. 4.7.6).

4.6 Approximate Max-Flow Algorithms

If we have limited resources, we want to be able to gracefully decrease the amount of computation needed without sacrificing much results quality. Thanks to the small expected diameter of small-world graphs, a huge number of short augment-

ing paths can be found in the first few rounds. According to our experiment results (see Section 4.7.8), most of augmenting paths are found in round $D/2$. Thus, cutting of the number of rounds to $D/2$ rounds may be good enough in approximating the max-flow value.

Another approximation strategy is to limit the excess paths length. We denote α as the maximum excess path length generated. The bigger the α, the more accurate the max-flow value and the longer the runtime and the bigger the number of rounds needed. Setting $\alpha >= D$ makes the algorithm to give the exact max-flow value. We evaluate the effectiveness of both approximation algorithms in Section 4.7.8.

4.7 Experiments on Large Social Networks

In all the experiments, we assume the graph to be bi-directional with each edge having a capacity of one. We used Hadoop version 0.21-RC-0 in a cluster of 21 machines connected with 1 Gigabit Ethernet. Each machine has 24 GB memory, 8-cores Hyper-threaded (2 × Intel E5520 @2.27GHz), 3 hard disks (@500GB SATA) and run Centos 5.4 (64-bit). One machine is used as the master node (also runs the *aug_proc*) and 20 machines as slave nodes.

In all of our experiments, we process the raw input graph in round #0 using MR to make the graph bi-directional and initialize unit edge capacities. For simplicity, unit capacities are used in the experiments but our algorithm supports rational numbers for the edge capacities. We modify some Hadoop configuration parameters such as: the number of maximum map and reduce tasks to 15 (up to 30 concurrent threads/node); the mapper/reducer memory limit to 640 MB; disable speculative execution for map and reduce (as it interferes with the schimmy optimization); DFS replication to 2; and vary the DFS block size (depending on the size of the graph). We did not do any other parameter tuning for the experiments.

4.7.1 FF1 Variants Effectiveness

FF1 is our initial design of the MR-based max-flow algorithm based on the Ford-Fulkerson method. As explained in Section 4.4.1, FF1 is composed of three composable techniques to improve the parallelism and to effectively reduce the number of rounds.

- $FF1(a)$ employs incremental updates to alter the residual network in parallel (in the next round) after an augmenting path is found and augmented.

- $FF1(b)$ employs bi-directional search which halves the number of rounds, doubles the amount of available parallelism by doubling the number of active vertices, and allows any vertex to generate augmenting paths which in turn allows many augmenting paths to be generated in one round yielding tremendous savings in number of rounds.

- $FF1(c)$ employs multiple excess paths which maintains the number of active vertices high throughout the rounds which gets more work done in each round and decrease the number of rounds further.

We evaluate these techniques by running each technique on $FB1$ graph. Note that $FF1(b)$ includes the $FF1(a)$ technique and $FF1(c)$ includes both $FF1(a)$ and $FF1(b)$ techniques.

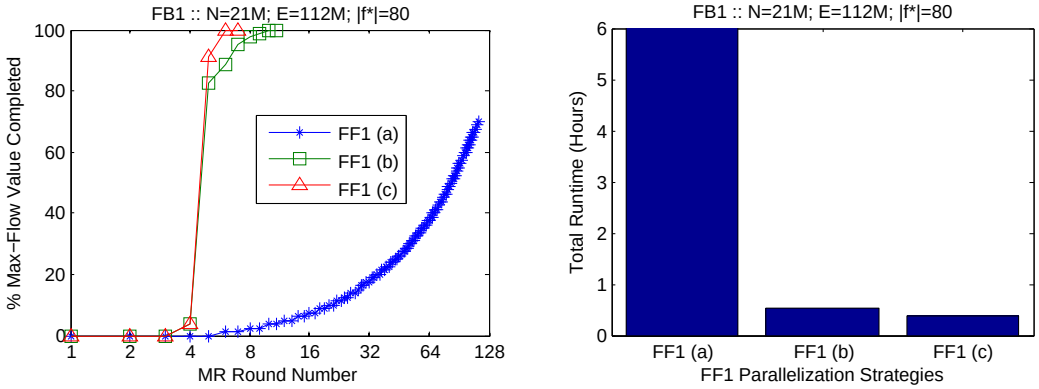


Figure 4.11: FF1 Variants on FB1 Graph with $|f^*| = 80$

Figure 4.11 shows one run of max-flow on $FB1$ on a random source s and sink t yielding a max-flow value of 80. The left graph shows the percentage of completion vs. the number of rounds. FF1 (a) spends more than 100 rounds and yet it has not complete the max-flow computation. FF1 (a) accepts around 1 augmenting path every 2 rounds. We suspect that there are a lot of congestion of excess paths that are being forwarded to t. We note that the naive translation of the Ford-Fulkerson method into MapReduce as described in Section 4.4 will accept 1 augmenting path every D rounds where D is the diameter of the graph. FF1 (b) frees the congestion by allowing any vertex in the graph to forward its augmenting paths directly to t. FF1 (b) accepts up to 63 augmenting paths in round 5 which gives a significant reduction in number of rounds needed. FF1 (c) further reduce the number of rounds and accept up to 70 augmenting paths in round 5. The right graph shows the total runtime in hours for each FF1 variants. Note that FF1 (a) is not yet complete. Since the runtime is proportional to the number of rounds, we see a tremendous savings in the FF1 (b) variant as it allows accepting a larger number of augmenting paths in a round. We observe that the

number of rounds required to compute max-flow on the Facebook graph is very small, close to the diameter of the graph. We will elaborate this in the following experiments on larger graphs.

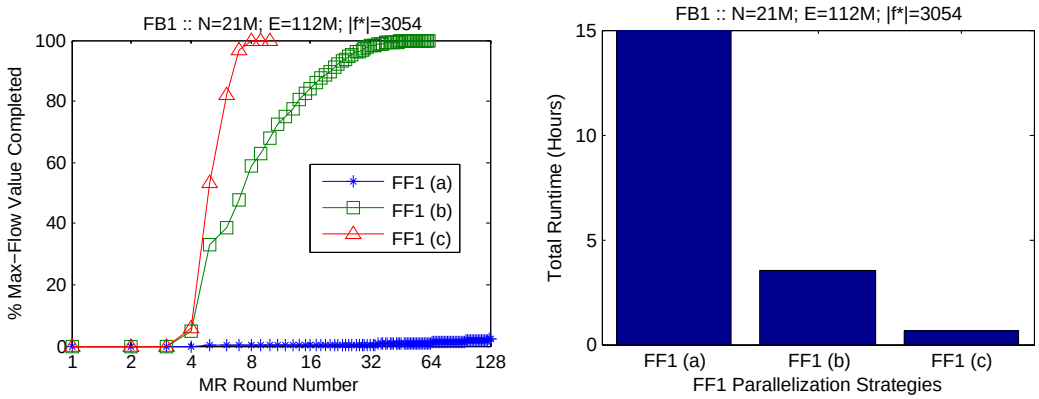


Figure 4.12: FF1 Variants on FB1 Graph with $|f^*| = 3054$

Figure 4.12 shows another run of max-flow on $FB1$ on a random source s and sink t yielding a large max-flow value of 3054. We see more dramatic differences among the FF1 variants. FF1 (a) is now performing very poorly as it accepts 1 augmenting path about every 2 rounds which is impractical as even after more than 100 rounds no significant progress has been made towards completion. FF1 (b) completes the max-flow computation in 63 rounds, thanks to the high number of augmenting paths accepted per round (up to 866). FF1 (c) cuts the number of rounds down further to only 10 rounds.

We investigate the effectiveness of the multiple excess paths optimization (FF1 (c) in Sec. 4.4.2) in reducing the number of rounds. The larger the number of excess paths stored in a vertex, the higher the available parallelism for finding augmenting paths as changes from augmented Δ flows is less likely to cause *all* excess paths in the vertex to be dropped which would make it inactive and not perform work for that round.

Fig. 4.13 shows the total runtime and the number of rounds required to compute the max-flow on the FB1 graph versus the number of excess paths stored (k). We see a significant drop in number of rounds as k increases. The results show a strong correlation between the number of rounds and runtime. An interesting situation happens when k gets larger. We might expect that as we increase the number of excess paths stored, it might be slower due to greater runtime overheads. However, the experiment shows that the number of rounds (and runtime) decreases as k increases even for large k. The overhead from storing more excess paths in a vertex is far smaller than the benefit of having fewer rounds.

In this section we have shown that each technique in the FF1 variants is crucial in designing a practical MR-based algorithm for small world network graphs.

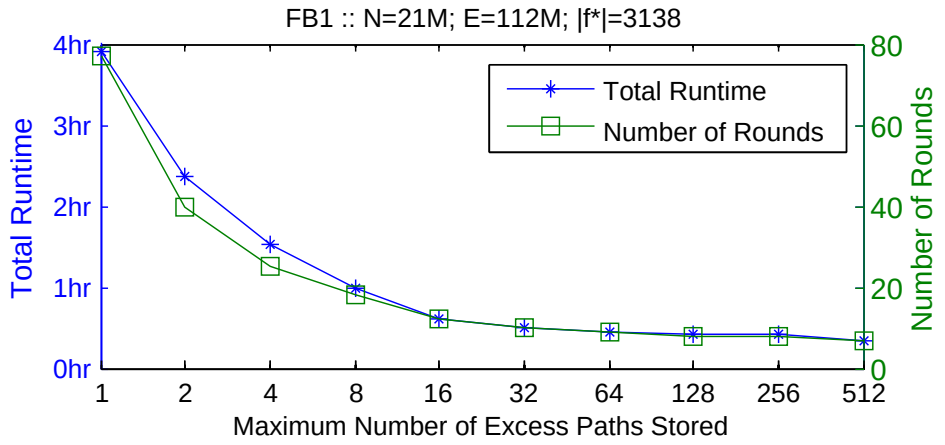


Figure 4.13: FF1 (c) Varying Excess Path Storage

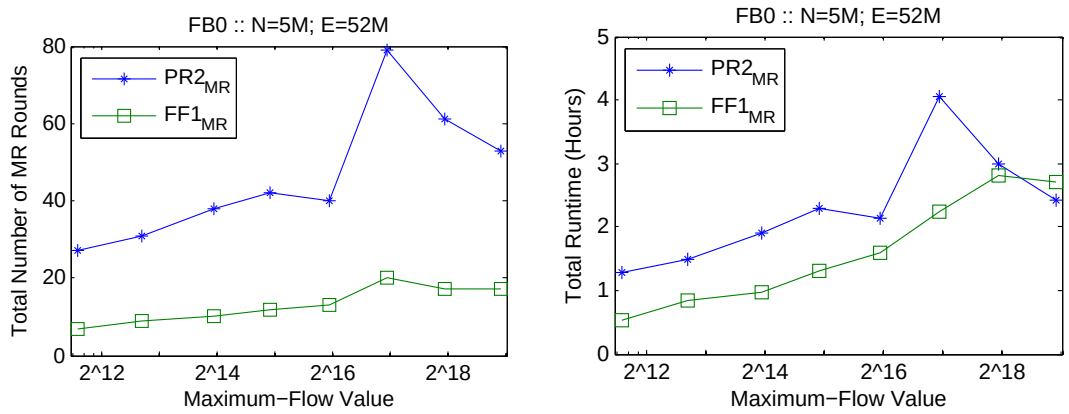


Figure 4.14: PR2_{MR} vs. FF_{MR} on the FB0 Graph

Experiments in the following sections will refer FF1 (c) as the FF1 variant as it combines all the techniques of FF1 (a) and FF1 (b).

4.7.2 FF1 vs. PR2_{MR}

In this section, we compare our FF1 algorithm with the *relaxed* PR_{MR} algorithm described in Section 4.3.5. We made an exception for this experiment that we follow the cluster setup as described in Section 4.3.6. In this experiment, we compare the number of rounds as well as the total runtime for FF1 algorithm and PR2_{MR} algorithm on computing max-flow on both $FB0$ and $FB1$. We do not compare against the PR_{MR} algorithm since it has been shown to perform less well than PR2_{MR} (see Section 4.3.6). We use the same s and t pairs for testing the flow scalability of PR2_{MR} in Section 4.3.6. There are 8 s and t pairs which represents tests with varying values for w from 1, 2, 4, 8, 16, 32, 64, and 128. The larger the w, the larger the max-flow value as there are more connections to the super source s and super sink t.

Figure 4.14 shows the comparisons on the $FB0$ graph. The left graph shows

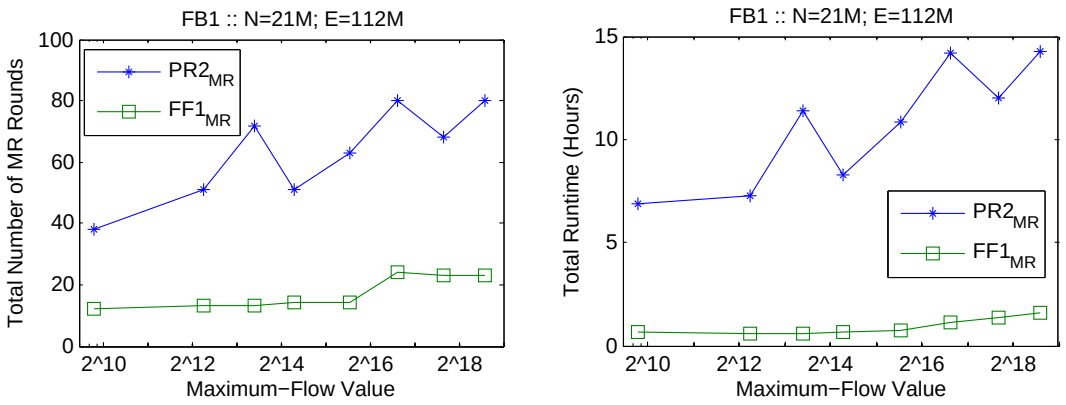


Figure 4.15: PR2_{MR} vs. FF_{MR} on FB1 Graph

the number of rounds vs. 8 runs in increasing max-flow values. FF1 consistently need a much smaller number of rounds across all max-flow values in the test. FF1 behaves more robustly as the spikes on certain max-flow values are far less sharp than that of PR2_{MR}. The FF1 runtimes are slightly faster but this becomes very much so with the next larger graph.

Figure 4.15 shows the comparison on the $FB1$ graph. We see a clear separation on both number of rounds and the total runtime. FF1 manages to keep the number of rounds small despite the large max-flow values. In Section 4.7.3 we show that this is still true for far larger graph size. PR2_{MR} algorithm is only practical for $FB1$-sized graph or smaller while FF_{MR} (and its optimized versions) is robust and far more scalable.

4.7.3 FF_{MR} Scalability in Large Max-Flow Values

This experiment tests the effect of increasing the max-flow value of our optimized FF_{MR} algorithm, FF5, on runtime and the number of rounds using the largest graph, FB6. Since each vertex in the Facebook graph has at most 5000 edges, this limits the upper bound on the max-flow value from s to t to be at most the minimum number of edges connected to both s and t. To create a much bigger max-flow value, we use the same approach described in Section 4.3.6, we select w random vertices that have a sufficiently large number of edges (at least 3000 edges) and connect them to a super source s. Similarly, we select another set of w vertices and connect them to a super sink t. The edge capacity from s and t to their connected vertices is set to infinity. To measure the effect of the max-flow value on runtime and the number of required rounds, we created several tests varying w. The larger the number of vertices w connected to s and t, the larger the potential max-flow value from s to t.

Figure 4.16 plots the runtime and rounds against the max-flow values for

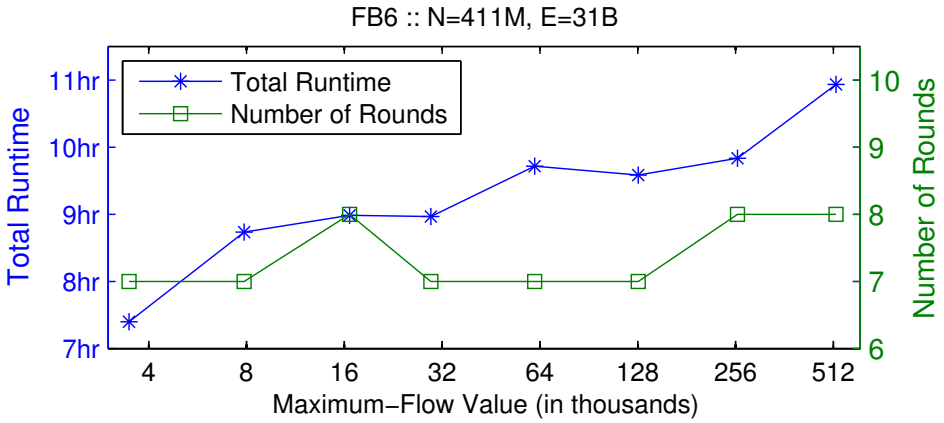


Figure 4.16: Runtime and Rounds versus Max-Flow Value (on FF5)

each w value from {1, 2, 4, 8, 16, 32, 64, 128}. We see that the total runtime only increases slowly with max-flow value $|f^*|$ as the x-axis is logarithmic. A surprising result is that the number of rounds required to complete the max-flow is relatively small – almost constant for any max-flow value tested. This result also suggests that the diameter D of the graph stays small despite the huge changes in the residual network due to large number of accepted augmenting paths. We believe this is due to the robustness of the small-world network. In practice, our algorithm design is effective in reducing the number of rounds close to D. The FF5 algorithm needs only as few as 8 rounds to compute max-flow even with a value as large as $|f^*| = 521551$. We estimate the value of D is between 7 to 14 for FB6 using a MR-based BFS from s. This experiment shows the feasibility of computing max-flow on very large small-world graphs with large max-flow values.

4.7.4 MapReduce optimization effectiveness

In Sec. 4.5, we introduce four further optimized versions of FF_{MR}. We show the effectiveness of the accumulated optimizations for each version in Fig. 4.17. Note that the runtime is in logarithmic scale and the number of rounds is labelled as 'R'. We tested the 5 versions of our FF_{MR} algorithms on two graphs: FB1 (smaller) and FB4 (bigger) to see the effectiveness of each version as the graph gets larger. FF1 is our baseline Ford-Fulkerson method on MR framework which already has many optimizations. The results for BFS (a simple graph traversal) are given as a comparison for a lower bound on the total runtimes.

The trend is that each successive algorithm reduces the runtime. FF2 gives ∼1.85× improvement over FF1 on FB1 graph by having an external process *aug_proc* specially to handle the acceptance of augmenting paths. The improvement becomes more significant (∼3.41×) with a larger graph (FB4) for reasons explained in Section 4.5.1. FF3 gives ∼1.25× improvement over FF2 on FB1 with

the schimmy design pattern. The improvement gets larger (∼1.74×) as the graph gets larger (on FB4). FF4 gives ∼1.16-1.41× improvement over FF3 by avoiding object instantiations. FF5 gives ∼1.68× improvement over FF4 by increasing the number of excess paths stored k up to the Facebook limit together with an optimization to prevent the re-sending of redundant excess paths in subsequent rounds. This reduces the number of rounds as well as overheads from record storage, shuffling and processing per round. The improvement is even bigger for larger graphs (∼2.07× for FB4). As the graph gets large, each one round of MR also has more overhead, thus a small reduction in number of rounds performed can give a significant runtime improvement.

Overall, the FF5 improvement is ∼5.43× faster than the original FF1 on the small FB1 graph and ∼14.22× on the larger FB4 graph.

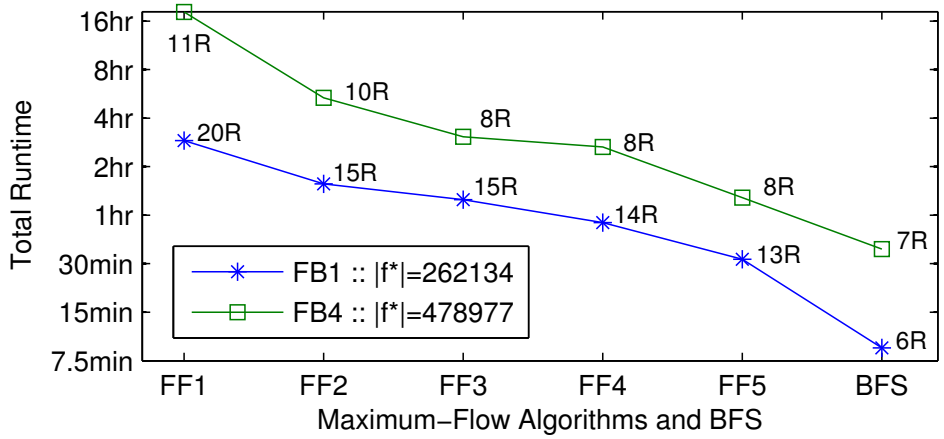


Figure 4.17: MR Optimization Runtimes: FF1 to FF5

4.7.5 The Number of Bytes Shuffled vs. Runtimes

Hadoop maintains internal counters during the execution of an MR job. In our FF_{MR} algorithms, whenever we run one MR job for each round and we can also get statistics of each round from the Hadoop internal counters. We are interested in particular counters that can give us information about the bottlenecks in MR, i.e., counters that correlate with the runtime. The external accumulator (*aug_proc*) process also maintains counters for the number of augmenting paths and the maximum processing queue size.

Table 4.5 shows the statistics of the FF5 algorithm on the FB6 graph with $w = 256$. FF5 requires 8 rounds (excluding round #0) to compute max-flow. Statistics for each round is given as a row in the table. The columns in the table are the round number (R), the number of augmenting paths accepted by *aug_proc* (A-Paths), the maximum size of the queue of *aug_proc* during the round (MaxQ), the number of intermediate records emitted by the mappers (Map Out), the

R	A-Paths	MaxQ	Map Out	Shuffle(KB)	Runtime
0	-	-	21,512 M	291,134,017	1:36:37
1	-	-	909	572	14:58
2	138,552	3	2 M	22,977	13:50
3	801,825	5,872	974 M	16,323,118	35:25
4	71,931	2,381	2,738 M	58,871,195	1:00:33
5	1,861	319	896 M	22,177,030	27:30
6	1	1	11,348 M	315,750,801	2:40:48
7	-	-	19,090 M	639,620,390	5:06:00
8	-	-	430 M	17,959,975	1:16:06

Table 4.5: Hadoop, *aug_proc* and Runtime Statistics on FF5

number of bytes shuffled across machines in kilobytes (Shuffle), and the running time of the round (Runtime).

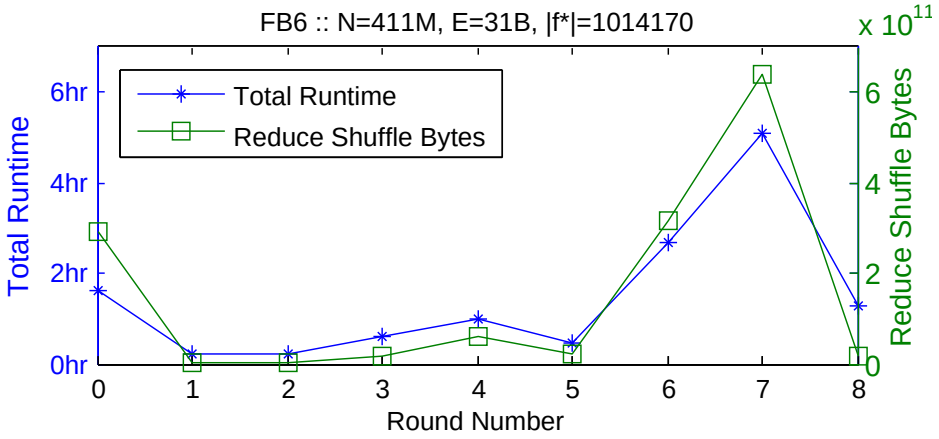


Figure 4.18: Reduce Shuffle Bytes and Total Runtime (FF5)

Figure 4.18 plots the $Shuffle$ and $Runtime$ from Table 4.5. We can see a strong correlation between the runtime and the number of shuffled bytes. The Shuffle column shows a large amount of data being shuffled between machines. The result is what we would expect – the more the intermediate records and size of data shuffled, the more the runtime. There is an approximately linear relationship (graph not shown) between Shuffle and runtime. In round #0, each vertex sends a message to each of its neighbors to establish bi-directional edge, hence the number of shuffled bytes is very large, but the size of the record is smallest. Round #1 has the smallest number of intermediate records and only a tiny fraction of the graph being changed (vertices s, t and their neighbors). Thus $\sim$15 minutes spent in round #1 is mostly due to MR overheads. In round #6 and #7, the number of bytes shuffled gets large because excess paths are being expanded very rapidly to visit new vertices.

Augmenting paths are found as early as round #2 (see the A-Paths col-

umn). The augmenting paths found are sent remotely to the processing queue in *aug_proc*, increasing the queue size. A thread in the *aug_proc* consumes the queue, decreasing the queue size. The experiment shows that the maximum queue size (MaxQ) is at most several thousand at any round which is hardly a bottleneck, hence, speculative execution is successful in finding many augmenting paths.

4.7.6 Shuffled Bytes Reductions on FF$_{MR}$ Algorithms

As we saw in Section 4.7.5 that the payload size correlates with the runtime of an MR job. Thus, it is important to design the algorithm to emit as small payload as possible. We improve our baseline FF1 algorithm design towards this goal of minimizing the number of shuffled bytes. In FF2 we extract out augmenting path from getting shuffled to t by sending directly to the remote accumulator program. In FF3, we apply the schimmy design pattern [80] and we prevent re-transmitting redundant excess paths in FF5.

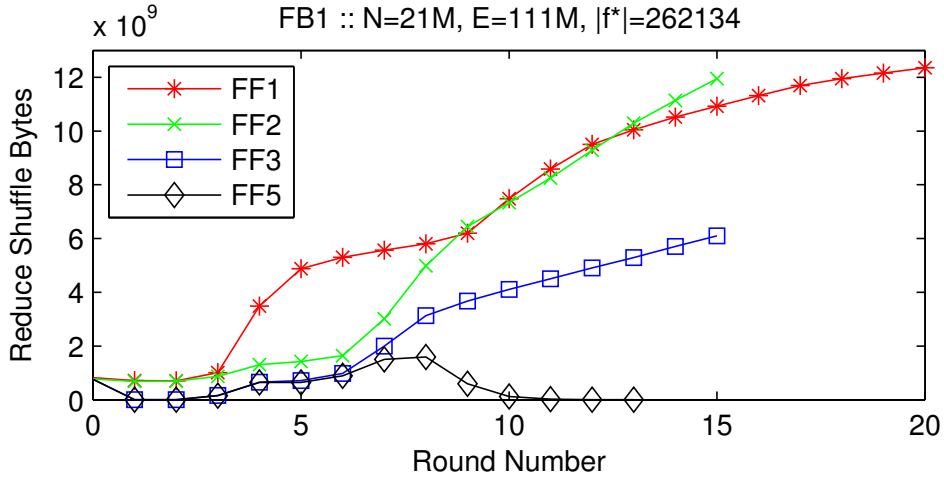


Figure 4.19: Total Shuffle Bytes in FF$_{MR}$ Algorithms

Figure 4.19 shows the reduction in the number of shuffled bytes on various versions of our FF$_{MR}$ algorithms. Each successive algorithm reduces the total shuffled bytes. FF2 has far fewer number of shuffled bytes compared to FF1 in round 3 to 9 because of the exclusion of augmenting paths from the intermediate records as they are sent directly to *aug_proc*. Since the storage of the augmenting paths is not handled by MR, the number of shuffle bytes becomes similar to FF1 again from round 10 onwards. FF3 has a consistently smaller number of shuffled bytes compared to FF2 due to the schimmy design pattern that prevents the master vertices from being shuffled. FF4 does not affect the number of shuffled bytes hence it is not shown. FF5 has far fewer number of shuffled bytes compared to FF3 after round 7 because it remembers the excess paths sent in the previous round and avoids re-sending them in subsequent rounds.

4.7.7 FF$_{MR}$ Scalability in Graph Size and Resources

We tested the scalability of our best FF$_{MR}$, FF5, against 6 different graph sizes (FB1 to FB6) as well as different number of slave nodes (5, 10, 20 slaves). For each graph size, the same random $w = 128$ vertices are used to have consistent results with different numbers of slave nodes. However, obviously the set of randomly selected w vertices need to be different for different graph sizes

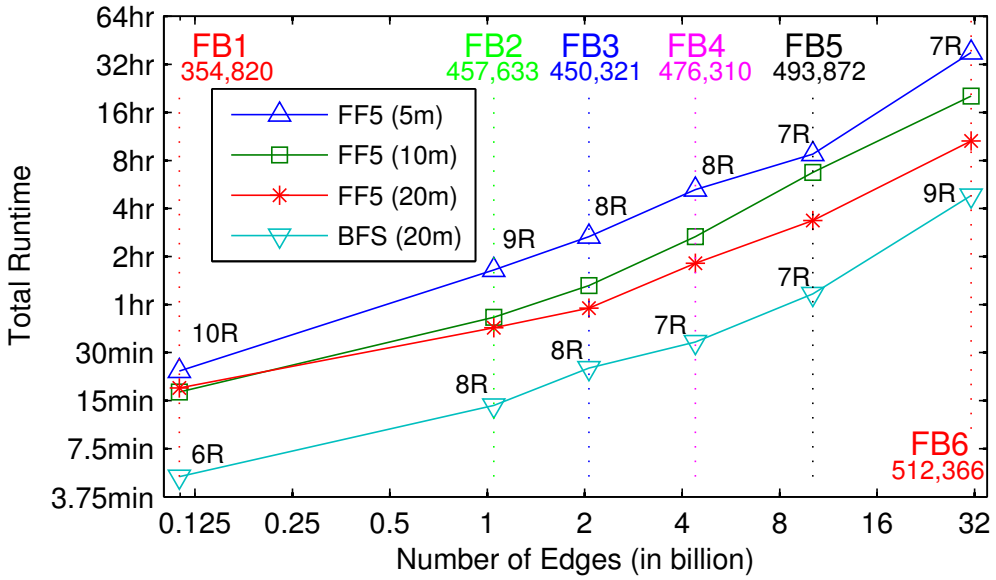


Figure 4.20: FF5 Scalability with Graph Size and Number of Machines

Figure 4.20 shows the scalability of our FF5 algorithm with different graph sizes, measured in terms of its number of edges. The maximum flow value for each graph FBi is given underneath each FBi label. Max-flow algorithms based on the Ford-Fulkerson method have complexity quadratic with respect to the graph size. Our FF$_{MR}$ algorithm, however, shows near linear runtimes with the graph size. We conjecture that this is due to the small world nature of the Facebook social network. This result suggests that complex graph algorithms can still be applicable to process very large small world network graphs. The figure also shows the linear scalability in terms of number of machines used (5, 10, 20 machines) across different graph sizes. We highlight that our FF$_{MR}$ algorithm is comparable in terms of number of rounds performed and only a constant factor (a few times) slower than the BFS algorithm in MR.

Figure 4.21 is another view of Figure 4.20 where the y-axis is the number of edges processed per second. The larger the graph size, the better our algorithm is in utilizing the slave machines. This suggests that the MR algorithm gives good scalability when the input size is large enough. Note that FB1 is a small graph (only 0.5G) – it is already at the lower limit for a MR job as it can be processed in-memory sequentially at one node. This also explains why there is

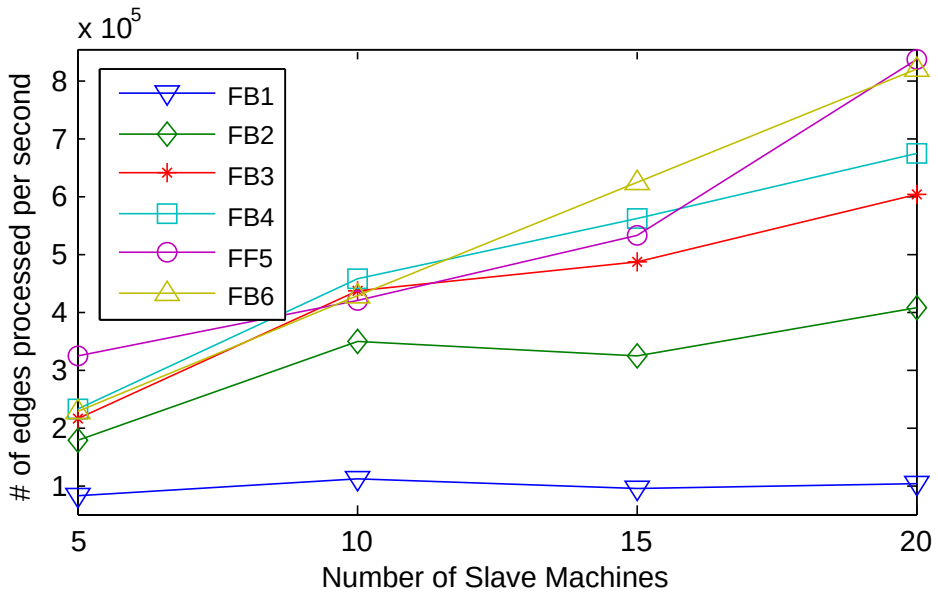


Figure 4.21: Edges processed per second vs. number of slaves (on FF5)

little differences on runtime for FB0 (see Section 4.7.2).

4.7.8 Approximation Algorithms

Table 4.5 shows that the last several rounds are the most expensive rounds in our algorithm and yet provides very little contribution in finding the max-flow values as in the last 3 rounds, there is little or no more augmenting paths accepted. The last 2 rounds are spent mainly on expanding excess paths to unvisited vertices to guarantee the optimal termination condition.

If we are willing to approximate the max-flow, the performance can be improved. The upper bound of a max-flow value can be computed as the sum of the capacity of the edges that are connected to the source vertex s (or t whichever is lesser). In the case of s is a super source, then the upper bound is the s neighbors' total edge capacity (similarly if t is a super sink). With the upperbound, we can vote to stop on the rounds when the max-flow value reaches a certain percentage from the upperbound.

Another alternative as mentioned in Section 4.6 is to limit the length of the excess path to α. We investigate both options in these experiments.

Figure 4.22 shows how the cut-off approximation algorithm perform. The left graph shows the runtime of the algorithm if it were cut-off at the n-th round. The right graph shows the percentage of max-flow value completed according to the computed upper bound. If we set the cut off to be at least 90% of the upper bound, then the algorithm will terminate on the 4th round in 49 minutes. If we set the cut off to be at least 60% of the upper bound, then the algorithm will

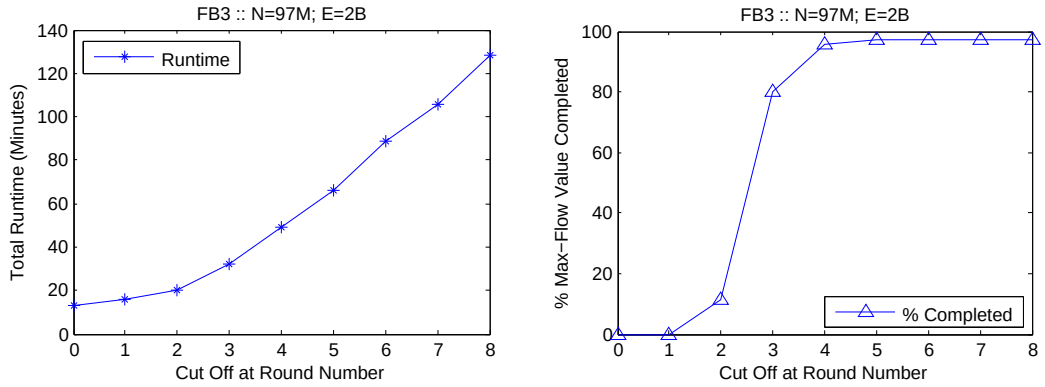


Figure 4.22: FF5 on FB3 Prematurely Cut-off at the n-th Round

terminate on the 3rd round in 32 minutes.

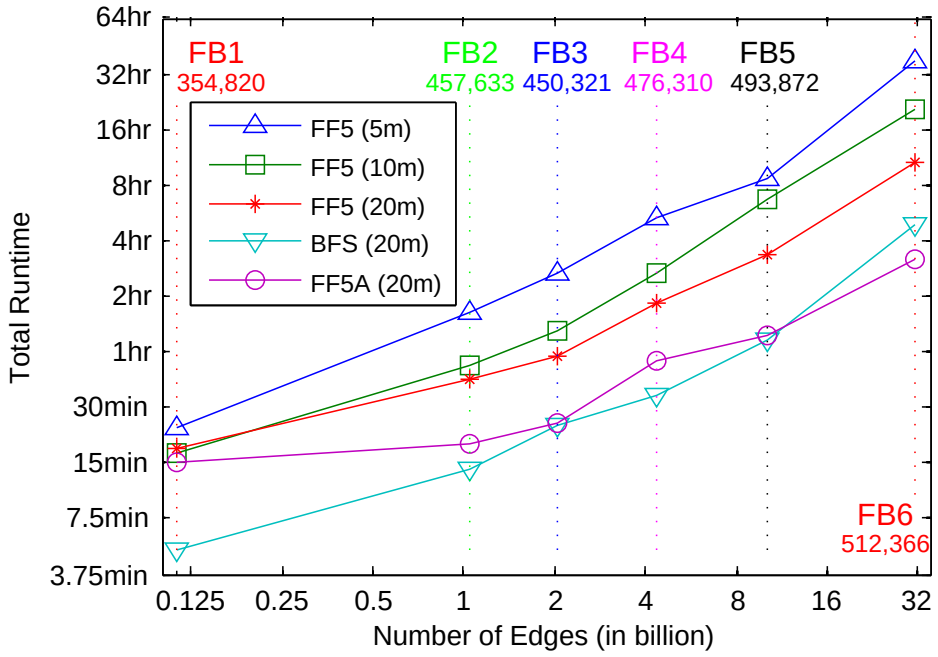


Figure 4.23: FF5A (Approximated Max-Flow)

Figure 4.23 shows FF5A algorithm that produces approximate max-flow value up to 90% from the upper bound. Due to the dense graph, the approximate max-flow algorithm, FF5A, finds more than 90% of the flows as early as 5 rounds thus giving performance comparable to that of BFS_{MR}.

Figure 4.24 shows the total runtime, number of rounds required, and the percentage of max-flow completed on different limit on the length of the excess path (α). We can see similar results that on 4th round, the max-flow is complete. The runtime trend is also similar with that of cut-off based approximation on the upper bound.

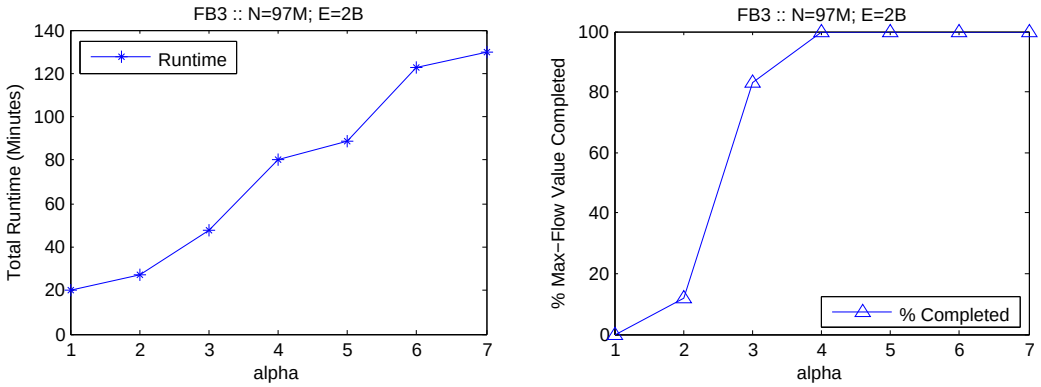


Figure 4.24: FF5 with varying α on the FB3 graph

4.8 Conclusion

In this chapter, we develop what we believe to be the first effective and practical max-flow algorithms for MapReduce. The algorithms employ techniques such as incremental updates, bi-directional search, and multiple excess paths that take advantage of the inherent properties of the graphs which allows it to run effectively and efficiently in practice.

While the best sequential max-flow algorithms have more than quadratic runtime complexity, we show it is still possible to compute max-flow efficiently and effectively on very large real-world graphs with billions of edges. We achieve this by designing FF_{MR} algorithms that exploit the small diameter property of such real-world graphs while providing large amounts of parallelism to scale well with the number of resources. We note that if the input graph is an arbitrary large graph (i.e., the graph does not have small world network properties), it may hit the worst case limit which might not be practically processable. Fortunately, we are not interested in such worst case graphs, rather the problem is to be solved for existing real world graph instances.

We identified bottlenecks in the system and present novel algorithm-system optimizations that significantly improve the initial design. These optimizations require understanding in both the algorithms and how MR works. The optimizations aim to minimize the number of rounds (which is the metric we use to evaluate our MR-algorithms complexity), the number of intermediate records to reduce network I/O overheads, and the size of the biggest record to reduce memory requirements to allow more workers to run in parallel.

Our experiments show a promising algorithm that scales linearly (in terms of graph size and number of machines) to compute max-flow for very large real-world sub-graphs such as those from Facebook.

Chapter 5

Conclusion

In this thesis, we study three problems that deal with big data. In this chapter, we present three important lessons that we learned in designing efficient and effective algorithms for big data problems. In all the algorithms, we maintain near-linear or sub-linear runtime complexity in order to scale to large data size with the rationale that the number of available resources can only be increased linearly as the free lunch is over [103]. Also, we must ensure our algorithms are robust across various datasets and workloads to consistently maintain the (sub)linear resource consumption. The first lesson is the importance of introducing stochastic behavior in the algorithm. The second is the exploitation of the inherent properties of the data being processed. The third is that the knowledge of both the system framework and the algorithms that run on top of it are important for optimizations.

5.1 The Power of Stochasticity

In chapter 2, we present our GDY algorithm which produces better segmentation quality than the heuristics (MHIST and MaxDiff). The GDY algorithm is based on stochastic local search (this is unusual since stochastic local search usually used in NP-hard problems). It starts with a random solution and continuously moves greedily to another solution until it stuck in a local optima. Thus, if we were to re-run the GDY algorithm again (using the same input), it may produce a slightly different result with similar quality. This stochasticity allows us to use the GDY algorithm as a sampling algorithm that consistently generates good sample solutions. In contrast, the heuristics MHIST and MaxDiff are deterministic which will produce the exact same result everytime it run (using the same input) which makes it unsuitable as a sampling algorithm.

The various solutions produced by multiple runs of GDY can be harnessed further by recombining those (already good) solutions to form the final solution.

In this sense, it is similar to a Genetic Algorithm that recombines bits and pieces from the solutions of a population to produce a new and better set of solutions. Fortunately, there already exists an optimal (albeit quadratic) segmentation algorithm which can be used in conjunction with GDY. Therefore, we can use the optimal segmentation to recombine the solutions produced by multiple runs of GDY into a significantly far better solution. In our experimental results, running several dozens of GDY runs is enough to produce a population which can be recombined into the optimal solution. Thus, we can effectively find solutions that are very close or match the optimal solutions in linear time $O(nB)$ rather than quadratic $O(n^2B)$ (using the optimal segmentation algorithm).

The role of the stochastic behavior in our segmentation algorithm is to consistently produce *good* solutions that can be recombined into significantly better solutions which outperform existing segmentation algorithms in both quality and performance.

In chapter 3, we expose the vulnerability of the database cracking philosophy that *do just enough*. We show that this philosophy fails under dynamic and unpredictable user query workloads. To do just enough, database cracking exclusively process the user queries by optimizing the physical datastore and cracker indexes that are strictly relevant to the queries. That is, it does not try to optimize the regions that are not touched by a query. While this brings a very lightweight adaptation to the user queries, it may severely penalize future queries because of bias in the user queries. Thus the original cracking may fail to adapt to user queries if the queries follow certain workloads. To mitigate this robustness issue, we introduce stochastic crack(s) for each user query. This ensures that no matter what the query workloads imposed by the user queries, the database cracking will introduce its own stochastic cracks to maintain the performance and quick adaptations to the future queries.

In chapter 4, a form of stochasticity (or non-determinism) allows processing large number of items in a streaming fashion and run concurrently with the MR job execution. In the FF2 variant 4.5.1, when augmenting paths are found in the middle of an MR job, they are immediately sent to the external accumulator process. The augmenting paths may arrive in any order. The external accumulator, immediately augment the paths as they arrive (i.e., first in first serve) in a streaming fashion. If an incoming augmenting path conflicts with the currently accepted augmenting paths, it will be rejected (discarded) otherwise it will be merged to the accepted augmenting paths. It is possible, however, to wait until all augmenting paths are received and then optimally select the maximum number of augmenting paths to be accepted. The downside is that it will require significantly larger memory resources and become a system bottleneck as it is not

run in parallel with the MR job (i.e., it is run after the MR job completes).

5.2 Exploit the Inherent Properties of the Data

In chapter 2, our stochastic local search algorithm, GDY, effectively captures the characteristics of the data sequence. The local search quickly finds segmentation positions that near or matches the optimal segmentation positions. Each run of GDY consistently discovers more than 50% of the optimal segmentation positions. Thus, within a few dozen runs of GDY, it manages to find almost all if not all optimal segmentation positions. When GDY_DP recombines the segmentation positions found by running GDY for several iterations, it produces a final segmentation that near if not matches the total error of the optimal segmentation.

In chapter 3, we showed that the original cracking has a robustness problem due to its philosophy that always *do just enough*. That is, it optimizes the physical data stores that are strictly relevant to the user queries without looking at the current properties of the data. To solve the robustness problem, we relaxed the cracking philosophy to also consider the current data property, in particular the piece size, and do more by performing stochastic cracks on the pieces that are relevant to the queries. Stochastic cracks are performed on pieces which sizes are more than the L1 cache size. This effectively reduces the robustness problem down to negligible levels.

In chapter 4, we are dealing with a seemingly intractable situation where the size of the data is so large and the existing algorithms have quadratic running time. Renting thousands of machines may not help in practically solving this big data problem since the amount of resources required can far outpace the amount of available resources.

In chapter 4, we give an example problem of computing a Maximum-Flow (max-flow) value in a large small-world network graph. The current state of the art max-flow algorithm is at least quadratic to the number of vertices in the graph and the graphs that arise from the Internet (such as online social network and the World Wide Web) can easily reach hundred millions vertices or more. In this situation, it may be tempting to resort to approximation algorithms which give approximate results. However, we discovered that we can exploit the inherent property of large real-world graphs to design a much more practical algorithm without sacrificing the result quality.

Most large real-world graphs have been shown to exhibit small-world network properties. In particular, they have a small diameter (i.e., the expected shortest distance between any two vertices in the graph is small). With the small diameter property, we can redesign a general purpose max-flow algorithm that have

quadratic runtime in terms of number of vertices into a very much linear runtime in terms of number of vertices and linear in terms of the expected diameter. Nevertheless, the robustness of this algorithm depends highly on the robustness of the graph itself. That is, since our algorithm alters the graph structure as the algorithm progresses, it is crucial that the expected diameter stays small even after large number of edge removals. We showed empirically that the online social network graph found in the Internet such as the Facebook graph is robust enough and we can practically compute max-flow in such graph effectively.

5.3 Optimizations on System and Algorithms

In chapter 4, we develop our max-flow algorithm on top of a distributed system framework called the MapReduce framework. It turns out that in order to optimize the overall processing, one needs a deep understanding in both the system framework and the algorithm that runs on top of it. The same algorithm can be tweaked to run an order magnitude faster by examining the systems/frameworks bottlenecks. The tweaks require deep understanding of both the algorithm and the framework limitations.

For example, in Section 4.5.4, we were trying to minimize the number of intermediate records being shuffled across machines since it is the biggest bottleneck of the MapReduce framework. We did this by adding an additional flag variable in our data structure to tell whether an excess path has been extended to which neighbor so that in the next MapReduce rounds, it can avoid re-sending unnecessary excess paths to that neighbor. In exchange, it requires each vertex to monitor all its extended excess paths for saturation and re-send new excess paths appropriately to the correct neighbors. Therefore, we made a tradeoff between no monitoring and re-sending excess path each round vs. monitoring saturated excess paths that have been extended and re-send as necessary. We discover that a significant amount of network resources can be saved by using monitoring and selective excess path extensions. Another example is that a significant disk I/O resources can be saved by employing the Schimmy method that avoids shuffling the master vertices as detailed in Section 4.5.2.

In the MapReduce framework, if one of record has very large size, it may create a load balance problem where all workers have long finished except the unlucky one that is processing the record with very large size. Unfortunately, in our case, we cannot split the record into two with smaller sizes since the record need to be processed atomically. As detailed in Section 4.5.1, we solved this load balance problem from a system improvement perspective by processing the record with very large size in a separate, dedicated stateful process.

Bibliography

[1] Amazon EC2 Pricing.
http://aws.amazon.com/ec2/#pricing.

[2] Apache Giraph.
http://incubator.apache.org/giraph/.

[3] Data Scientist Study.
http://marketaire.com/2012/01/17/demand-for-data-scientists/.

[4] Facebook Statistics.
http://www.facebook.com/press/info.php?statistics.

[5] Graph 500.
http://www.graph500.org.

[6] Graph 500.
http://www.oracle.com/technology/events/hpc_consortium2010/graph500oraclehpcconsortium052910.pdf.

[7] Hadoop.
http://hadoop.apache.org.

[8] Hadoop Combiners.
http://developer.yahoo.com/blogs/hadoop/posts/2010/08/apache_hadoop_best_practices_a/.

[9] Mapreduce and Hadoop Algorithms in Academic Papers.
http://atbrox.com/2011/11/09/mapreduce-hadoop-algorithms-in-academic-papers-5th-update.

[10] Serge Abiteboul, Rakesh Agrawal, Phil Bernstein, Mike Carey, Stefano Ceri, Bruce Croft, David DeWitt, Mike Franklin, Hector Garcia Molina, Dieter Gawlick, Jim Gray, Laura Haas, Alon Halevy, Joe Hellerstein, Yannis Ioannidis, Martin Kersten, Michael Pazzani, Mike Lesk, David Maier, Jeff Naughton, Hans Schek, Timos Sellis, Avi Silberschatz, Mike Stonebraker,

Rick Snodgrass, Jeff Ullman, Gerhard Weikum, Jennifer Widom, and Stan Zdonik. The lowell database research self-assessment. *Communications of the ACM*, 48:111–118, May 2005.

[11] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1999. ACM Press.

[12] R. Albert, H. Jeong, and A.L. Barabasi. The diameter of the world wide web. In *Nature*, 1999.

[13] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. *ACM CoRR Computing Research Repository*, abs/1111.4570, 2011.

[14] David.A Bader and Vipin Sachdeva. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. In *International Conference on Parallel and Distributed Computing Systems*, 2005.

[15] Richard Bellman. On the approximation of curves by line segments using dynamic programming. *Communications of the ACM*, 4(6):284, 1961.

[16] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Linear time bounds for median computations. In *ACM Symposium on Theory of computing*, pages 119–124, 1972.

[17] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Conference on Innovative Data Systems Research*, pages 225–237, 2005.

[18] Nicolas Bruno and Surajit Chaudhuri. To tune or not to tune? a lightweight physical design alerter. In *International Conference on Very Large Data Bases*, pages 499–510, 2006.

[19] Nicolas Bruno and Surajit Chaudhuri. An online approach to physical design tuning. In *International Conference on Data Engineering*, pages 826–835, 2007.

[20] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *VLDB Journal*, 10(2-3):199–223, 2001.

[21] Kaushik Chakrabarti, Eamonn Keogh, Sharad Mehrotra, and Michael Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *ACM Transactions on Database Systems*, 27(2):188–228, 2002.

[22] R. Chen, X. Weng, B. He, and M. Yang. Large graph processing in the cloud. In *ACM SIGMOD International Conference on Management of Data*, 2010.

[23] Boris V. Cherkassky and Andrew V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1994.

[24] Gianmarco De Francisci Morales, Aristides Gionis, and Mauro Sozio. Social content matching in mapreduce. *International Conference on Very Large Data Bases*, 4:460–469, April 2011.

[25] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, 2004.

[26] E.A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Dokl.*, 1970.

[27] John Douceur. The sybil attack. In *1st International Workshop on Peer-to-Peer Systems*, pages 251–260, 2002.

[28] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. In *Journal of the ACM*, 1972.

[29] Gary William Flake, Steve Lawrence, and C. Lee Giles. Efficient identification of web communities. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 150–160, 2000.

[30] L.R. Ford and D.R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, pages 399–404, 1956.

[31] Minos Garofalakis and Amit Kumar. Wavelet synopses for general error metrics. *ACM Transactions on Database Systems*, 30(4):888–928, 2005.

[32] S. Ghemawat, H. Gobioff, and S.T. Leung. The google file system. In *Symposium on Operating Systems Principles*, 2003.

[33] Phillip B. Gibbons, Yossi Matias, and Viswanath Poosala. Fast incremental maintenance of approximate histograms. *ACM Transactions on Database Systems*, 27(3):261–298, 2002.

[34] Anna C. Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *ACM Symposium on Theory of computing*, New York, NY, USA, 2002. ACM Press.

[35] Andrew V. Goldberg. Recent developments in maximum flow algorithms. In *Scandinavian Workshop on Algorithm Theory*, 1998.

[36] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. In *IEEE FOCS Symposium on Foundations of Computer Science*, 1997.

[37] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. In *ACM Symposium on Theory of computing*, pages 136–146, New York, NY, USA, 1986. ACM.

[38] Goetz Graefe. Robust query processing. In *International Conference on Data Engineering*, page 1361, 2011.

[39] Goetz Graefe, Stratos Idreos, Harumi Kuno, and Stefan Manegold. Benchmarking adaptive indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 169–184, 2010.

[40] Goetz Graefe and Harumi Kuno. Adaptive indexing for relational keys. In *Self-Managing Database Systems*, pages 69–74, 2010.

[41] Goetz Graefe and Harumi Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Conference on Extending Database Technology*, pages 371–381, 2010.

[42] Sudipto Guha. On the space-time of optimal, approximate and streaming algorithms for synopsis construction problems. *VLDB Journal*, 17(6):1509–1535, 2008.

[43] Sudipto Guha and Boulos Harb. Approximation algorithms for wavelet transform coding of data streams. *IEEE Transactions on Information Theory*, 54(2):811–830, 2008.

[44] Sudipto Guha, Nick Koudas, and Kyuseok Shim. Approximation and streaming algorithms for histogram construction problems. *ACM Transactions on Database Systems*, 31(1):396–438, 2006.

[45] Sudipto Guha, Kyuseok Shim, and Jungchul Woo. REHIST: Relative error histogram construction algorithms. In *International Conference on Very Large Data Bases*, pages 300–311, 2004.

[46] Felix Halim, Stratos Idreos, Panagiotis Karras, and Roland H.C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. In *International Conference on Very Large Data Bases*, 2012.

[47] Felix Halim, Panagiotis Karras, and Roland H.C. Yap. Fast and effective histogram construction. In *ACM Conference on Information and Knowledge Management*, 2009.

[48] Felix Halim, Panagiotis Karras, and Roland H.C. Yap. Local search in histogram construction. In *AAAI Conference on Artificial Intelligence*, 2010.

[49] Felix Halim, Roland H.C. Yap, and Yongzheng Wu. A mapreduce-based maximum-flow algorithm for large small-world network graphs. In *International Conference on Distributed Computing Systems*, 2011.

[50] Steven Halim and Roland H.C. Yap. Designing and tuning sls through animation and graphics: an extended walk-through. In *Engineering Stochastic Local Search Algorithms*, pages 16–30, 2007.

[51] Steven Halim, Roland H.C. Yap, and Hoong C Lau. Viz: A visual analysis suite for explaining local search behavior. In *ACM Symposium on User Interface Software and Technology*, pages 57–66. ACM Press, 2006.

[52] Johan Himberg, Kalle Korpiaho, Heikki Mannila, Johanna Tikanmäki, and Hannu Toivonen. Time series segmentation for context recognition in mobile devices. In *IEEE International Conference on Data Mining*, Washington, DC, USA, 2001. IEEE Computer Society.

[53] Stratos Idreos. *Database Cracking: Towards Auto-tuning Database Kernels.* PhD thesis, Centrum Wiskunde en Informatica, 2010.

[54] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Conference on Innovative Data Systems Research*, pages 68–78, 2007.

[55] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Updating a cracked database. In *ACM SIGMOD International Conference on Management of Data*, pages 413–424, 2007.

[56] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column stores. In *ACM SIGMOD International Conference on Management of Data*, pages 297–308, 2009.

[57] Stratos Idreos, Stefan Manegold, Harumi Kuno, and Goetz Graefe. Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. *International Conference on Very Large Data Bases*, 4(9):585–597, 2011.

[58] N. Imafuji and M. Kitsuregawa. Finding web communities by maximum flow algorithm using well-assigned edge capacities. In *IEICE Transactions on Information and Systems*, 2004.

[59] Yannis E. Ioannidis. Universality of serial histograms. In *International Conference on Very Large Data Bases*, 1993.

[60] Yannis E. Ioannidis. Approximations in database systems. In *International Conference on Database Theory*, 2003.

[61] Yannis E. Ioannidis. The history of histograms (abridged). In *International Conference on Very Large Data Bases*, 2003.

[62] Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *ACM SIGMOD International Conference on Management of Data*, pages 233–244, 1995.

[63] Yannis E. Ioannidis and Viswanath Poosala. Histogram-based approximation of set-valued query-answers. In *International Conference on Very Large Data Bases*, pages 174–185, 1999.

[64] M. Ivanova, N. Nes, R. Goncalves, and M. Kersten. Monetdb/sql meets skyserver: the challenges of a scientific database. *Scientific and Statistical Database Management Conference*, 2007.

[65] H. V. Jagadish, Nick Koudas, S. Muthukrishnan, Viswanath Poosala, Kenneth C. Sevcik, and Torsten Suel. Optimal histograms with quality guarantees. In *International Conference on Very Large Data Bases*, pages 275–286, 1998.

[66] U. Kang, Spiros Papadimitriou, Jimeng Sun, and Hanghang Tong. Centralities in large networks: Algorithms and observations. In *SIAM International Conference on Data Mining*, pages 119–130, 2011.

[67] U Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Fast diameter estimation and mining in massive graphs with hadoop. *Technical Report CMU-ML-08-117*, 2008.

[68] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Symposium on Discrete Algorithms*, 2010.

[69] Panagiotis Karras. Multiplicative synopses for relative-error metrics. In *Conference on Extending Database Technology*, New York, NY, USA, 2009. ACM.

[70] Panagiotis Karras. Optimality and scalability in lattice histogram construction. In *International Conference on Very Large Data Bases*, 2009.

[71] Panagiotis Karras and Nikos Mamoulis. One-pass wavelet synopses for maximum-error metrics. In *International Conference on Very Large Data Bases*, 2005.

[72] Panagiotis Karras and Nikos Mamoulis. The Haar$^+$ tree: a refined synopsis data structure. In *International Conference on Data Engineering*, Washington, DC, USA, 2007. IEEE Computer Society.

[73] Panagiotis Karras and Nikos Mamoulis. Hierarchical synopses with optimal error guarantees. *ACM Transactions on Database Systems*, 33(3):1–53, 2008.

[74] Panagiotis Karras and Nikos Mamoulis. Lattice histograms: a resilient synopsis structure. In *International Conference on Data Engineering*, 2008.

[75] Panagiotis Karras, Dimitris Sacharidis, and Nikos Mamoulis. Exploiting duality in summarization with deterministic guarantees. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 2007. ACM Press.

[76] Robert Kooi. *The Optimization of Queries in Relational Databases*. PhD thesis, Case Western Reserve University Cleveland, 1980.

[77] M. Kulkarni, M. Burtscher, R.Inkulu, K.Pingali, and C.Cascaval. How much parallelism is there in irregular applications? In *Symposium on Principles and Practice of Parallel Programming*, 2009.

[78] Kristen LeFevre, David J. DeWitt, and Raghu Ramakrishnan. Mondrian multidimensional k-Anonymity. In *International Conference on Data Engineering*, Washington, DC, USA, 2006. IEEE Computer Society.

[79] Wentian Li. Dna segmentation as a model selection process. In *Proceedings of the fifth annual international conference on Computational biology*, RECOMB '01, pages 204–210, New York, NY, USA, 2001. ACM.

[80] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Mining and Learning with Graphs Workshop*, 2010.

[81] Martin Lühring, Kai-Uwe Sattler, Karsten Schmidt, and Eike Schallehn. Autonomous management of soft indexes. In *Self-Managing Database Systems*, pages 450–458, 2007.

[82] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *ACM Symposium on Principles of Distributed Computing*, 2009.

[83] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *VLDB Journal*, 9(3):231–246, 2000.

[84] Yossi Matias, Jeffrey Scott Vitter, and Min Wang. Wavelet-based histograms for selectivity estimation. In *ACM SIGMOD International Conference on Management of Data*, 1998.

[85] Stanley Milgram. The small world problem. In *Psychology Today*, 1967.

[86] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *ACM/USENIX Internet Measurement Conference*, 2007.

[87] M. Muralikrishna and David J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1988. ACM Press.

[88] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[89] Gregory Piatetsky-Shapiro and Charles Connell. Accurate estimation of the number of tuples satisfying a condition. In *ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1984. ACM Press.

[90] Viswanath Poosala, Venkatesh Ganti, and Yannis E. Ioannidis. Approximate query answering using histograms. *IEEE Data Engineering Bulletin*, 22(4):5–14, 1999.

[91] Viswanath Poosala and Yannis E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *International Conference on Very Large Data Bases*, pages 486–495, 1997.

[92] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *ACM SIGMOD International Conference on Management of Data*, pages 294–305, 1996.

[93] Frederick Reiss, Minos Garofalakis, and Joseph M. Hellerstein. Compact histograms for hierarchical identifiers. In *International Conference on Very Large Data Bases*. VLDB Endowment, 2006.

[94] H. Saito, M.Toyoda, M.Kitsuregawa, and K.Aihara. A large-scale study of link spam detection by graph algorithms. In *Adversarial Information Retrieval on the Web*, 2007.

[95] Marko Salmenkivi, Juha Kere, and Heikki Mannila. Genome segmentation using piecewise constant intensity models and reversible jump MCMC. In *European Conference on Computational Biology*, 2002.

[96] Oskar Sandberg. Distributed routing in small-world networks. In *Algorithm Engineering and Experiments*, 2006.

[97] K. Schnaitter et al. Colt: Continuous on-line database tuning. In *ACM SIGMOD International Conference on Management of Data*, pages 793–795, 2006.

[98] Y. Shiloach and U. Vishkin. An $o(n^2 \log n)$ parallel max-flow algorithm. In *Journal of Algorithms 3*, 1982.

[99] K. Shvachko, H. Kuang, S. Radia, and R.Chansler. The hadoop distributed file system. In *Storage Conference*, 2010.

[100] S. Spek, E. Postma, and H.J.V.D. Herik. Wikipedia: organisation from a bottom-up approach. In *Wikisym*, 2006.

[101] Michael Stonebraker. One size fits all: An idea whose time has come and gone. In *International Conference on Data Engineering*, pages 869–870, 2005.

[102] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: a column-oriented DBMS. In *International Conference on Very Large Data Bases*, pages 553–564, 2005.

[103] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 2005.

[104] Evimaria Terzi and Panayiotis Tsaparas. Efficient algorithms for sequence segmentation. In *SIAM International Conference on Data Mining*, 2006.

[105] Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian. Sybil-resilient online content voting. In *Symposium on Networked System Design and Implementation*, 2009.

[106] Jeffrey Scott Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *ACM SIGMOD International Conference on Management of Data*, 1999.

[107] Haifeng Yu, Michael Kaminsky, Phillip B. Gibbons, and Abraham Flaxman. Sybilguard: Defending against sybil attacks via social networks. In *ACM Special Interest Group on Data Communication*, pages 267–278. ACM Press, 2006.