Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

Adaptive Compression for Graph Processing

Author:
Rob Moore

Supervisor:
Dr. Holger Pirk

Second Marker:
Prof. Peter Pietzuch

June 17, 2018

Abstract

Many graph-modeled data sources are subject to fast-changing, unpredictable workloads, such as a news story being searched for as it happens, or a trending post or person on social media causing an influx of related, similar queries. Most processing frameworks for graph data sources contain preprocessing steps to speed up the overall analysis, which prohibits them from responding to queries until that step is done. In this report we explore potential solutions to this problem by building novel variations of database cracking, an algorithm designed for auto-tuning relational databases, and showing how the cracking algorithm can be changed to improve performance for graph data processing.

Acknowledgements

I would like to thank my supervisor Holger Pirk for his knowledge and direction during the project. I am also grateful to my personal tutor, Paul Kelly, for his advice about projects in general and about optimization and graph algorithms, as well as all of the interesting discussions we've had while I've been at Imperial. Finally I'd like to thank my dad who scanned the boundary pieces of this report and gave me some advice on how to improve it.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Connected Data Sources

Data sources featuring connections between entities are frequently used in the modern era to represent abstract networks, most notably, the world wide web and social networks. The nodes and edges of these graphs are analyzed using graph algorithms in order to learn something about the network, such as a which websites are trending or which individuals are most followed.

1.2 Modeling Graph Data

To store a graph in computer memory, we represent it as an array of edges. Each element of the array contains a source vertex and a destination vertex, which we denote as *src* as *dst*. This is appropriately described as an *edge array* format.

An advancement on an *edge array* is to cluster one side of the edges by vertex. For example all of the out-neighbours of every vertex might be clustered together. We might even, rather than store an *edge array*, use a direct mapping from each vertex to an array containing its out-neighbours[1]. This storage format for a graph is called an *adjacency list*, and converting an *edge array* into an *adjacency list* is a common preprocessing step for many graph processing frameworks.

If we want to also store data specifically about vertices, we can also store vertex data separately in a key-value store of some kind, mapping vertex id to vertex data. A frequent approach to this is to map the given vertex ids onto the integers counting up from 0, and to store the ith vertex' data in the ith element of a vertex data array.

1.3 Graph Indexing

To speed up queries on graph data, we can build indices, just like we can with relational databases. These are structures based on the data which enable us to perform queries faster. For example, the preprocessing that converts an edge array into an adjacency list can be considered a form of indexing, that is, an adjacency list is an index for an edge-array. There are a few terms used to describe different kinds of indexing, below we define three important ones.

- **Offline indexing** is the term used to describe building indices before queries are introduced to a system.
- **Online indexing** describes a technique in which the workload and performance of a system is examined and then the required indices are built when needed.
- **Adaptive indexing** is where an index is built in the course of answering incoming queries. The index is partially built as a collateral effect of answering queries.

[1] This of course also can be done with in-neighbours.

1.4 Database Cracking

Database cracking is an adaptive indexing technique originally created for relational databases. The index it builds towards is a sorted column. It does this by applying pivoted partitioning while scanning during range queries. Columns begin with values in an arbitrary order, but every range query causes values to get reorganized such that after the query is finished, all values are pivoted around the selected range.

1.5 Adaptive Graph Processing

The biggest names in the technology industry; Google, Facebook, Twitter etc. all rely on connected data. The functionality their services support is frequently subject to rapidly changing and unpredictable workloads in the forms of new trends, new news stories and new people. In the modern era, adapting to unpredictable query workloads is important in processing graph data. Towards this end we have explored novel techniques for adaptively indexing graph data.

1.6 Objectives

In this project, we have studied the potential of compression-based variations of database cracking when applied as an adaptive indexing technique for graph data. Compression is an obvious dimension with which to vary the original cracking algorithm; the algorithms we created effectively perform the conversion from edge array into adjacency list adaptively, and in the case of run-length encoding, values are compressed at even lower levels of granularity within the edge array.

1.7 Contributions

- We introduce a range of compression-based variations of the cracking [5] algorithm.
- We present a novel variant of the cracking algorithm which outperforms the original on the benchmarks we measured.
- We demonstrate two implementations of cracking variants which run-length encode the column while scanning, and evaluate them against the original cracking algorithm.

1.8 Report Outline

Section 2 discusses related work in the fields of graph processing, graph indexing and workload-aware processing frameworks. In section 3, we discuss the necessary preliminary background for our contributions. This constitutes an explanation of the original cracking algorithm. We describe and explain our contributions in section 4, and evaluate them in section 5. We conclude the report in section 6 with a summary of what we did and future work that can be undertaken to build upon our work.

Chapter 2

Related Work

2.1 Workload-Aware Frameworks

2.1.1 Database Cracking

From Idreos et al. [5], cracking is an auto-tuning technique for relational databases. A column is indexed by being having a copy of it physically restructured in-place during queries, such that it is partially sorted.

Figure 2.1 shows two queries being run against a column within a system employing database cracking. We can see that Q1 copies the original column into a cracker column. The cracker column is then scanned to retrieve the result tuples. As a side effect of the scan, the cracker column is partitioned around the pivots of the range query, and after returning, the new information about the structure of the column is stored to be exploited later. Q2 again scans the column, while also applying pivoted partitioning. The new information gained from the scans is shown in the diagram and could be exploited later in order to reduce the number of tuples that needed to be scanned.

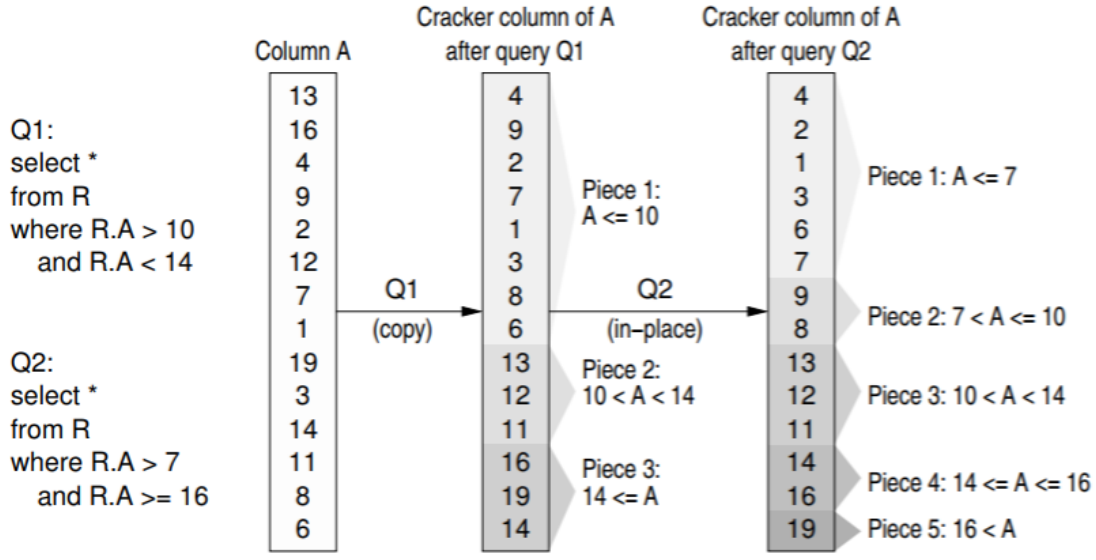

Figure 2.1: Example of a column being cracked

The CPU efficiency of standard cracking has been improved since the original algorithm [8]. This was done by first identifying the causes of CPU inefficiency in the algorithm, which were shown to be bad speculation and instructions entering the pipeline at the frontend. These problems were solved by first applying predication to cracking in order to remove the possibility for wasted cycles caused by bad speculation. The authors then identified the performance bottlenecks for the predicated implementation and mitigated them using vectorization.

In the same paper, a parallel implementation of cracking was introduced. The initial parallelization of the algorithm was essentially the same as the parallelization of a sequential scan into a parallel scan. This was then refined to produce a more performant parallel implementation.

The workload-robustness of database cracking has been improved by a technique called Stochastic Cracking [3]. This technique aims to improve the *workload-robustness* of cracking. It identifies a weakness of cracking: That it can experience reduced performance when given a non-ideal workload, in which it may rescan the same values many times, such as in the case of a sequential workload, in which every query requests a range of values that follow the previous one. The solution presented in this paper was to create algorithms which were not purely query driven in their decisions to reorganize sections of the column, but also used the column data and randomness. They presented a range of techniques, and showed that they solved the lack of workload-robustness in standard cracking, while also mostly preserving the lightweight, adaptive properties of it for ideal workloads.

Standard cracking is discussed in greater detail in chapter 3.

2.1.2 Group-by-Query

In-part inspired by cracking was the thesis work of Aluç [1], who proposed a group-by-query (G-by-Q) representation for RDF data, for which the structure of individual database records, as well as the way records are serialized on the storage system are dynamically determined based on the workload.

Using G-by-Q, the way database records are serialized and their contents are determined dynamically by the workload. The format of database records is determined by the queries on the database. The diagram below shows an example of the storage for a database which has had two types of queries applied to it. P_1 to P_3 are the results of a linear query, whereas P_4 and P_5 are both the results of star-shaped queries.

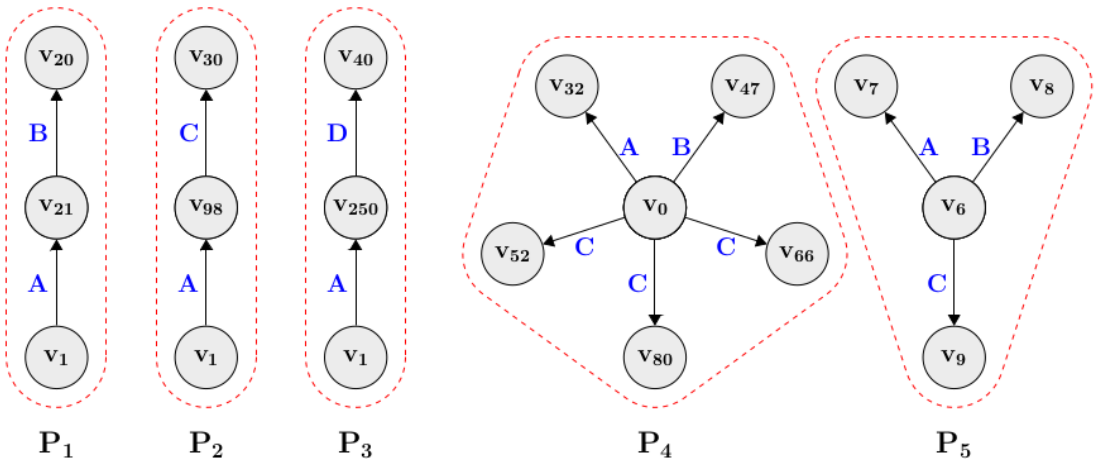

Figure 2.2: Example of G-by-Q storage representation

This technique proved to be fast and robust against other popular frameworks for querying RDF data, however, the system is complicated - Aluç's implementation was reported to be over 35,000 lines of C++. In this work we aimed to produce simpler contributions. Additionally, the G-by-Q technique is used for point-type queries, which seek to find all instances of certain subgraphs within a graph based on node labels, whereas our contributions are focused on traversal queries, which move through the graph by following edges and making computations based off the node and edge properties.

2.2 Graph Processing

2.2.1 Ligra Framework

Ligra [9] is a lightweight graph processing framework for shared-memory multi-core machines for graph traversal algorithms, such as pagerank and BFS. Ligra takes the form of a simple API of three routines: size, edgeMap and vertexMap, as described in the paper.

size(U) returns the number of vertices in the set U.

edgeMap(G, U, F, C) applies the function F to all edges in G with source vertex in U and target vertex satisfying C.

vertexMap(U, F) applies the function F to every vertex in the set U.

A crucial advantage of the Ligra framework is that in the cases of both edgeMap and vertexMap, the supplied function can run in parallel, however, the user must ensure parallel correctness.

2.2.2 Frequency Based Clustering

Frequency based clustering [10] constitutes physically reorganizing the vertex data such that frequently accessed vertices are clustered together. In the case of traversal algorithms, as were studied specifically by this paper, the property by which to cluster vertices is their degree (in- or out-degree depending on the algorithm). By doing this clustering, cache contention between threads is reduced thanks to the improved locality between frequently accessed vertices. This improves cache utilization and reduces the cycles spent stalled on memory. The authors found that real world graphs often exhibit inherent locality, and disturbing the structure of the vertex data too much causes performance to worsen. They determined that they achieved the best performance when they clustered together vertices at one end of the graph only if their degree exceeded the mean degree across all nodes.

2.2.3 CSR segmenting

This technique, introduced in the same paper [10] as frequency based clustering, aims to optimize the cache performance of a graph algorithm by making random accesses go only to the cache and by making all memory accesses sequential. It does this by working only on a single cache-sized segment of the vertex data at a time. By segmenting the graph into cache-sized subgraphs, all the required vertex data for the processing of a single subgraph can be stored in the cache.

This method requires preprocessing of the adjacency list, followed by the processing of each segment. Segments are processed one at a time, but the computation within the segment is fully parallelized. There is no cache-contention because the threads all share the same read-only working set (vertex data). We can avoid high merging cost after the computations by not using too many segments. The author's did experiments using a high number of segments which fit in the L2 cache, however, the best performance was achieved by using fewer segments, each of which fit in the LLC and contained a large number of edges. After the segments are all processed a low-cost cache-aware merge is used to combine the intermediate results from each segment.

2.2.4 Space filling curve layouts

An alternative to clustering an edge array representation of a graph by vertex is to cluster edges according to the coordinates of a space filling curve. This has the potential to provide locality in both the source and destination vertices.

The most famous example of this is the Hilbert Curve, which is a continuous mapping between a number and a point on a 2D square. Numbers which are close by are mapped closely together - in this way the Hilbert ordering of an index of an adjacency list is somewhat locality preserving. We can use a Hilbert ordering in order to improve locality on edge accesses, which therefore leads to improved cache performance [6].

Chapter 3

Background

3.1 Database Cracking

Database cracking was invented by Stratos Idreos as a method of auto-tuning for database kernels by using workload-aware physical restructuring on queried columns. It forms the basis for our contributions in which we apply variants of cracking to graph algorithms with adaptive compression.

3.1.1 Cracker Column

To create an adaptive index for a relational database using a given column, we first copy that column into what is called a cracker column. When executing queries on the chosen column, we execute those queries on the cracker column using modified database operations called cracking operations. In this report we only need to use selection, however cracking can be exploited for insertion, deletion, updates and joins, as well as other operations. We will denote the cracker column by *crk*.

3.1.2 Cracker Select

Initially, there is no information known about the values in the column and their locations. When a ordering query is sent to the cracker column, the column is scanned, and after the scan, all values too small to be selected appear contiguously at the start of the column, followed by all of the selected values, followed by all of the values too large to be selected. Below is an illustration indicating the effect of a cracking operation.

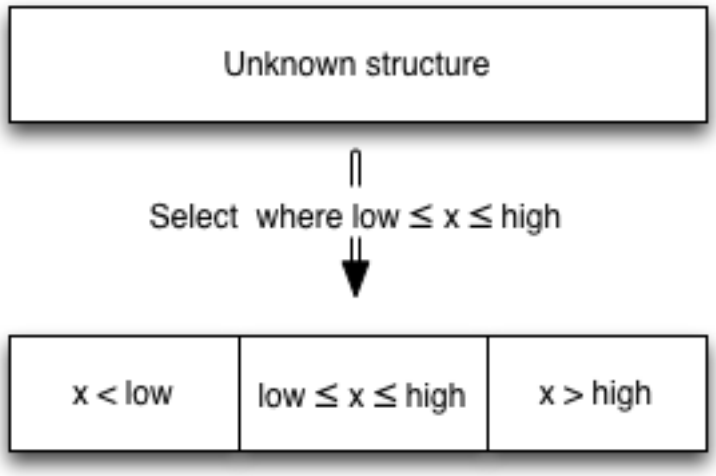

Figure 3.1: Cracking physically restructures the column towards being sorted

Every time we perform a new cracking scan, we learn more information about the values within the column and their associated locations. Our aim is to store and exploit this information to achieve a speed up in performing queries. Notably, selections which occur first in queries are clustered first. In this way the physical layout of the cracker column is adaptive to the workload the database is under. The contiguous regions created through cracking operations are called column fragments.

3.1.3 Column Fragments

Central to the use of cracking is the concept of a column fragment. This represents a contiguous region of the cracker column whose constituent values are potentially bounded by known values. Initially, the entire column represents the only column fragment where there are no bounds yet on the values. However, as queries come in, the scans across the cracker column cause us to gain knowledge about contiguous regions of the column and the values which bound them. As we build up more knowledge about the column, the column fragments get broken up smaller and smaller, meaning that the size of our scans is decreasing as well, given that we are exploiting our knowledge about the fragments and their bounds.

The information about the column fragments is stored in the cracker index.

3.1.4 Cracker Index

The cracker index is a storage data-structure which holds the workload-aware memory about the cracker column fragments. It is denoted *CrkIdx* and is subject to one invariant:

$$\texttt{(value, index)} \; \epsilon \; \texttt{CrkIdx} \implies \forall i < \texttt{index} : crk[i] < \texttt{value}$$

In plain English, if a `(value, index)` pair are stored in the cracker index, then all values before `index` in the cracker column are less than `value`.

When performing queries, we want to scan only the necessary parts of the column, so we use the information stored in the cracker index to identify the smallest column fragment which is known to contain all of the values we are looking for. After executing such a scan, we put information into the cracker index, effectively breaking up the column fragment we just scanned so that future queries will have to scan less.

3.1.5 Tuple Reconstruction

The cracker column is a copy of the original column, however, the cracking operations only apply to the cracker column, so to reconstruct original tuples based on our results from the cracker column, we use an additional array to act as the mapping between indices in the cracker column and indices in the base (original) columns. This is *late* tuple reconstruction. When the cracker column is initialized, this array is also initialized, its value being an enumeration of the indices from 0 to the length of the column. We call this array the *base_index*. All restructuring operations that happen to the cracker column also happen to the *base_index*, in order to maintain the mapping between cracker and base columns.

3.1.6 Crack-in-three Algorithm

The algorithm we describe here as "the cracking algorithm" is actually the "crack in three" variant of the algorithm as described by Idreos. We are using this because for our use case, in which we are always selecting a single node, the crack-in-three algorithm is appropriate, whereas the simpler and faster crack-in-two scan is appropriate when querying for values one side of a given value.

The cracker select operator is called with a low value, *low* and a high value, *high*, as well as two booleans which indicate the inclusivity of these boundaries. The select returns all the values between *low* and *high* in the column using the specified inclusivities.

The cracking algorithm can be grouped into five stages: Setup, tighten, scan, memo and return.

Stage 1: Setup

In the first stage of the algorithm, a contiguous section of the column is selected for scanning. If the column hasn't been cracked yet, the cracker column and *base_index* are initialized.

Using the arguments, we define the smallest and largest values in the selected range, which depend on the inclusivity of the range at each end.

$$\sigma_{min} = \begin{cases} \text{low} + 1^a\texttt{, if not inclusive} \\ \text{low}\texttt{, if inclusive} \end{cases}$$

$$\sigma_{max} = \begin{cases} \text{high - 1}\texttt{, if not inclusive} \\ \text{high}\texttt{, if inclusive} \end{cases}$$

[a]For integers, the minimum difference between elements is 1, however, for other data types, this may be different. This is discussed further in 4.2.1.

We then determine the initial values for the two *edge pointers*. There is a low pointer and a high pointer, which we will denote as L and H. These two pointers are subject during the scan to the following invariants.

$(L1)\ \forall i < L : crk[i] < \sigma_{min}$
$(L2)\ crk[L] \geq \sigma_{min}$
$(H1)\ \forall i > H : crk[i] > \sigma_{max}$
$(H2)\ crk[H] \leq \sigma_{max}$

The edge pointers then, bound the region of the cracker column in which all of the sought values lie. It would be optimal to initialize them as *tightly* as possible. When we say *tightly*, what we mean is as far from their associated edge as possible. For L this means as close to the end of the column as possible, and for H this means as close to the head of the column as possible.

To initialize the edge pointers, we search in the cracker index for values which most tightly bound the region of cracker column we have to scan. For L, we are looking for the value in the cracker index which is as high as possible, but no more than *low*. If no such value exists, L is initialized at 0. Similarly, for H, we seek the value in the cracker index which is as low as possible, but no less than *high*. If no such value exists, H is initialized as $|crk| - 1$.

Stage 2: Tighten

The tightening operation involves moving the two edge pointers inwards as far as possible while maintaining their invariants. In practice this is a while loop which checks the associated invariant and if it still holds, tightens the pointer (increment in the case of L, decrement in the case of H).

During this stage, it is possible to discover that no results exist. For example, if we try to select a value greater than any value in the column, then L will advance all the way off the tail end. For H of course, this happens when we select a value lower than any in the column, with it decrementing off the start of the column. In both of these cases, we return empty results.

Both tightening operations (low-side and high-side) are used in the next stage of the algorithm as well.

Stage 3: Scan

At the start of this stage, L is duplicated to produce the *iteration pointer*, denoted I, which is subject to a single invariant.

$$\forall i : L \leq i < I, \sigma_{min} \leq crk[i] \leq \sigma_{max}$$

The iteration pointer is used to scan the fragment from L to H. As I encounters values within the fragment, it is determined where in the final arrangement they should lie. For example, if the value is less than σ_{min}, then it is swapped to the value at L, and then L is tightened. This maintains the L invariant, while making progress towards the situation in which only selected values are between L and H, which is the point at which the scan in finished.

If the value at I is one of the sought values, then I is advanced, which maintains its invariant. Because of this, when I surpasses H, this means that all values between L and H inclusive are in the selected range, and by the invariants on L and H, every value in the selected range in the column lies between L and H, meaning that we're finished with the scan.

Figures 3.2 to 3.4 show the low side swap case from the identification of the swap to the advancement of the low pointer.

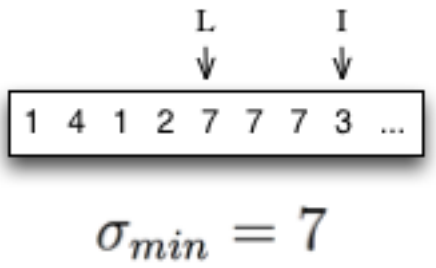

Figure 3.2: Standard Cracking: Low-side swap recognized

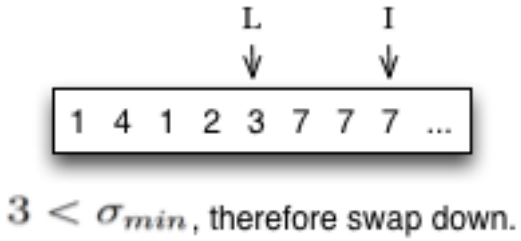

Figure 3.3: Standard Cracking: Low-side swap performed

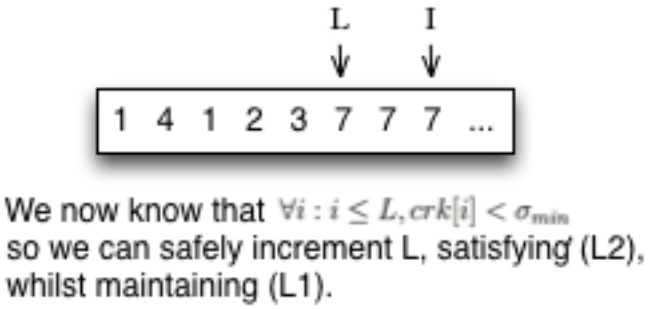

Figure 3.4: Standard Cracking: Low edge pointer tightened after swap

Figures 3.5 to 3.7 show the same thing for the high-side swap.

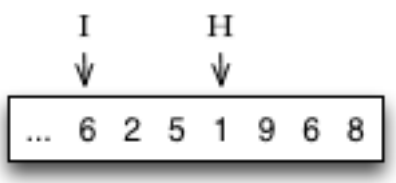

Figure 3.5: Standard Cracking: High-side swap recognized

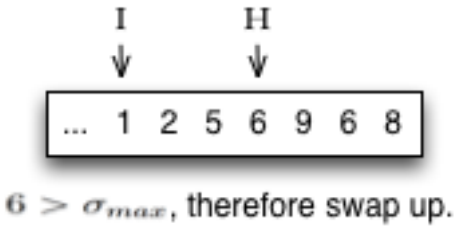

Figure 3.6: Standard Cracking: High-side swap performed

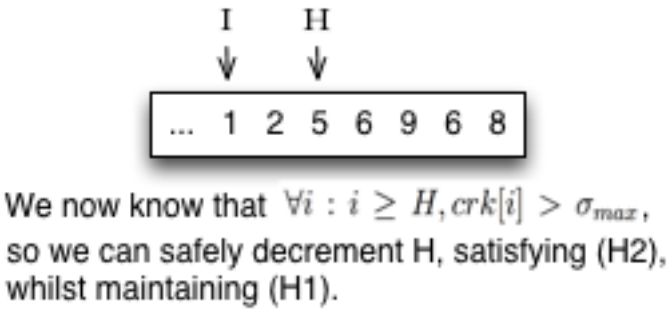

Figure 3.7: Standard Cracking: High edge pointer tightened after swap

Stage 4: Memo

At this point, we know for a fact that all values before L in crk are also less than σ_{min}. This combination of L and σ_{min} is enough information to satisfy the main property of the cracker index, so we insert (L, σ_{min}). Additionally, we know all values before and including H in crk are less than σ_{max}, so all values before $H+1$ in crk satisfy the required cracker index property. Hence we also insert $(H+1, \sigma_{max})$.

Stage 5: Return

Having acquired the range of cracker column indices in which the selected values lie, we map this back to indices of the original columns using the `base_idx` array and retrieve values from the desired column.

Chapter 4

Adaptive Compression

4.1 Cracking

In this chapter we examine ways of performing online compression of the adjacency list representing the graph by modifying the existing cracking algorithms presented by Ideos' et al.. The background subsection on cracking covers the terms we will use in this chapter to explain the modifications we made.

Our motivation for applying compression to cracking is to be able to run algorithms faster by exploiting uniform sections of the cracker column.

In this chapter, we will first discuss how we recognize opportunities to compress contiguous, uniform values within the cracker column. Then we will discuss the ways in which the information required to repeatedly exploit these uniform sections of the column can be stored. In the relevant sections and subsections we will describe our implementation of various techniques, which we see the evaluation of the next chapter.

4.2 Opportunities

Given an adjacency list which we are querying using cracking, we can gather information about uniform column-sections with varying eagerness in the course of a cracking scan. We can reserve a decision on compression until we know that an entire column-fragment is compressible, or we can make many small compressions while scanning a fragment of the cracker column.

When we compress only after identifying an entirely uniform column fragment, we call this "per-fragment" compression, because we are compressing entire column-fragments only.

Applying compression during the course of a scan is "eager" compression. Within the space of possibilities for eager compression, we have only explored run-length encoding (RLE) in this project.

4.2.1 Per-fragment

When applying per-fragment compression, we update the cracker index with the boundary values for any created fragment(s) after performing the column scan. If two stored indices are *minimally different*, then we can compress the column in that range, which corresponds to a uniform column-fragment. Minimal difference between stored values is important to per-fragment compression for this reason, and later in this subsection we will explain the concept in more detail.

Advantageously, this method is very easy to implement. One must simply check the boundaries of newly created fragments and determine if any fragments can now be known to be uniform. The cost of this technique on top of normal cracking is a branch in which the cracker index is checked to see if the column fragment under consideration is uniform, meaning the selection can return early.

Minimal Difference

We use the phrase *minimally different* to describe two values for which there exists no value between them in the range of values that can be taken for their common type. For example, if we consider

the integers, the values 2 and 3 are minimally different, because there exists no integer between these values. For a data-type such as a floating point number, this value will vary depending on the accuracy with which the user wishes to cluster - it may be acceptable to the application that the values 2.33334 and 2.33279 both be considered 2.33, however, unlike in the integer case, there is information being lost during compression, which may not be acceptable to the user. In this case they may opt to use an appropriately fine granularity for the compression according to their application.

4.2.2 Run-length Encoding

Our aim with run length encoded cracking is to be able to speed up the scan by enabling scanning pointers to hop over long runs of the same value. For this, we build and maintain information about all runs of consecutive nodes in the cracker column. This is done inside an array called *run_lengths*. *run_lengths*[*i*] indicates to a scanning pointer that the next `i` values (inclusive) are the same, and therefore can be considered together, whether that means that they are hopped over, or swapped away to another part of the column.

Compaction with RLE presents no significant additional difficulties compared to compaction with per-fragment compression.

The cracker column is scanned both front to back and back to front, therefore it would be preferable that information be stored such that it can be applied in both scanning directions. For a given run of consecutive values in the cracker column from index i to index j inclusive, the *run_lengths* array stores the value $1 + j - i$ at both indices i and j, that is, the number of values in the consecutive run.

We also need to maintain the *run_lengths* array under the restructuring that takes place during cracking to make it worthwhile, which requires that we be be careful in maintaining all of the necessary invariants regarding the position of the two tightening edge pointers used during cracking.

This must also take account of the fact that two runs of different lengths may need to be swapped. When making these swaps of different length runs, the *run_lengths* array may have need to have its entries modified in order to retain correctness. In this chapter we discuss two potential approaches to swapping around runs of different lengths within the *run_lengths* array.

The first is to swap the entire of the shorter run and modify entries of the *run_lengths* array corresponding to the longer run to maintain consistency after the swap. This means that the longer run is not fully swapped, so we have called it "underswapping". Underswapping causes the longer run to get broken up into two runs, one run the same size as the smaller run, which gets swapped entirely with the smaller run, and the other, a run consisting of the entire remainder of the longer run.

The second approach is to swap the entire of the longer run, and pad the shorter run with more elements. This requires the padded elements to have their runs checked and potentially have entries modified, due to the possibility that they are constituents of another run. Due to the fact that the shorter run is swapped with more elements than are actually in that run (by padding), we call this "overswapping". Overswapping maintains runs after they are created, however it is much more complex.

4.3 Storage

After identifying a uniform column section, we have a choice in how to proceed. We can compact the cracker column by deleting duplicated values. Alternatively, we could store the information needed for us to exploit the compression opportunity, without making changing to the physically compacting the data.

4.3.1 Compaction

We describe the choice to delete duplicated values as "compaction". Our implementation uses per-fragment granularity. In our implementation, the cracker column is an array, meaning that to delete arbitrary values requires large copying of memory from the tail side of the array towards the front. Furthermore, when we delete values from the cracker column and copy part of the array

towards the head, any entries in the cracker index whose stored pointer will have been offset by the shift must be updated.

It must be possible to reconstruct deleted values after compaction. To solve this, we store information about compacted values, however, we do it indirectly. We store an array called offsets, denoted ofs, which holds for each cracker column index, its corresponding index in the base columns. Whenever a value is compressed, that value as well as its minimally different successor value are stored in the cracker index, so when making a selection, if there is only a single instance of the node being selected within the cracker column, we can check to see if it is compressed by comparing its entry in the offsets array with that of the following index. Any compressed value will have a non-consecutive index in its successor, indicating that once there were values between them, which have now been compacted, as illustrated by figure 4.1.

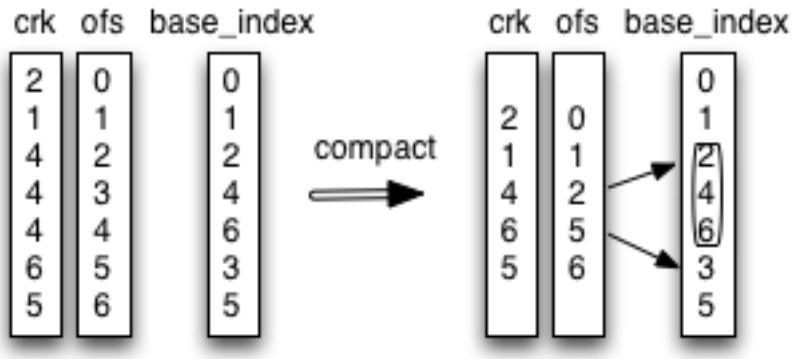

Figure 4.1: Information about compactions are held in the offset array

The potential advantages of using compaction are that we can fit more edges into memory, that we can improve cache utilization by improving spatial locality among edge clusters within the edge array.

The costs for applying compaction are copying of memory upon compaction, traversing the AVL tree of the cracker index to update all the shifted entries upon compaction, and the maintenance of the array of offsets.

Algorithm Overview

The format of the algorithm is almost the same as standard cracking. After a uniform column fragment is identified, that fragment is compacted. When returning a section of the column, if there are compacted values within the selection, they must be decompressed into the base index by using the offset array.

Figure 4.2 compacting duplicated values results in changes within the cracker index to maintain correctness.

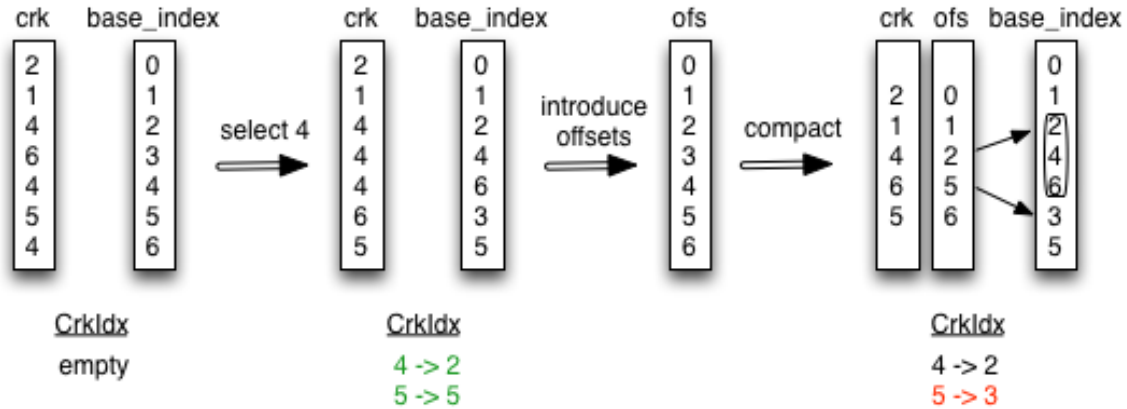

Figure 4.2: Compaction causes entries in the cracker index for greater values to be reduced

The algorithm starts the same as for standard cracking, however, there is the added introduction of the offset column, which is initialized to 0, because before any compactions have been made, elements in the cracker column are not offset from elements in the base columns. The tightening phase is identical to that of standard cracking.

After the tightening phase, if a single value is identified as the only selected value, then we must check for compression by checking the cracker index to see if this value has been selected (and therefore compressed) before. If so then we can return immediately by decompressing the value at that index in the cracker column.

During the scan, previously compacted values in the cracker column are never swapped around. In our case it is impossible because a previously compressed column fragment will never be contained within a larger range query since our queries are for ranges of just a single node id. Even in the general case, swapping compacted values can be avoided by only cracking the boundary pieces - that is, by not scanning column fragments which are known to be included in the final result anyway, and instead physically reorganizing only the first and last fragments in the selected range.

Single values in the cracker column are swapped around just like for standard cracking, however, the $base\_index$ array may have a different length to the cracker column due to compactions, so the way swaps happen is changed slightly. We use the properties of the offset array to ensure that the correct swaps take place, under the knowledge that no compacted values will ever be swapped.

$$\forall i : 0 \leq i < |crk| - 1,\ \forall j : ofs[i] \leq j < ofs[i+1],\ crk[i] = base\_column[base\_index[j]]$$
$$\forall j : ofs[|crk| - 1] \leq j < |base\_column|,\ crk[j] = base\_column[base\_index[|crk| - 1]]$$

With these properties in mind, we swap values in the base index found by passing the relevant pointers through the offset array first. That is, rather than swapping for example the elements at L and I, instead we swap the values at $ofs[L]$ and $ofs[I]$.

Immediately after the scan the cracker index is updated and checked for compression opportunities. If the cracker index contains minimally different values, then the values of the smaller run can be compressed, because we know the index in the cracker column before which all entries are exceeded by that value, and also the index after which all values exceed that value. This is the principle of recognition compression, which is discussed in more detail in the next subsection. Figure 4.3 illustrates this.

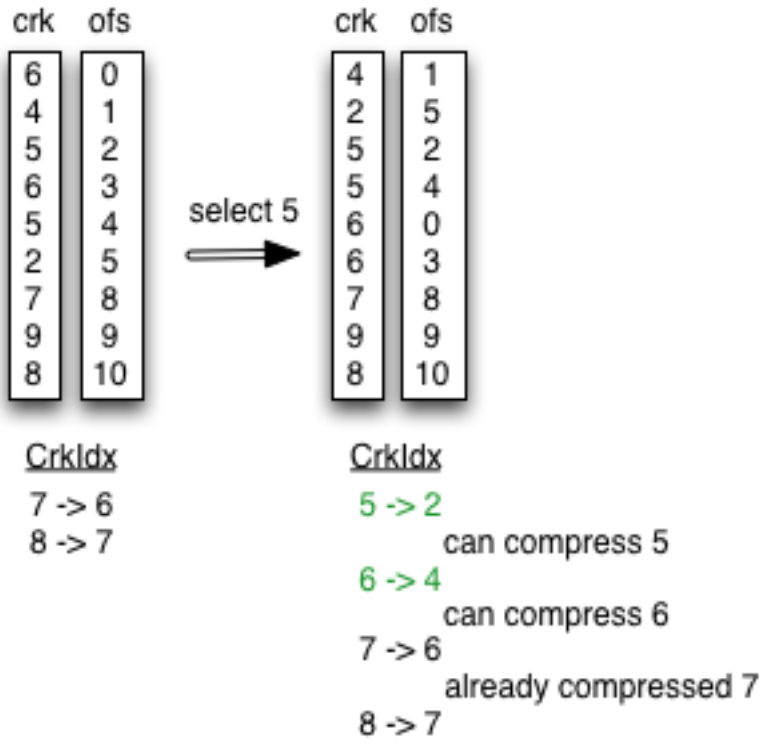

Figure 4.3: Minimally different values in the cracker index imply a compression opportunity

If a compression opportunity is found, the compaction is performed. This involves deleting the contiguous indices of the duplicated values for the cracker column and the offsets column. However, when we delete these values, the entries in the cracker column after the compacted value are shifted towards the head of the array. This means that their indices within the cracker column change, and therefore any entries in the cracker index to those values are incorrect. To amend this, all entries in the cracker index whose value is greater than the value being compacted have their index reduced by the number of values compacted. This is because entries in the cracker index with a greater value than the value being compacted necessarily appear later in the column and therefore are shifted during compaction. The reduction is by the number of values compacted, because this is the size of the shift and therefore the size of the required amendment to maintain correctness.

We must take care that the various potential compactions we can perform after the scan do not interfere with one another. Figure 4.4 indicates a scenario in which this can happen. In this diagram, a selection for the value 3 has just finished the scanning phase and is ready to go into compaction. By compacting the lower values first, the implementation must account for headward shifts in the cracker column's values, otherwise the insertion of the next value into the cracker index will be incorrect. In our solution, we address this problem by performing compactions in descending order of the value being compacted.

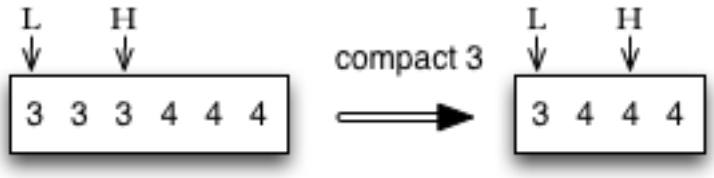

Figure 4.4: Compactions can affect each other, headward shifts in later sections of the array must be accounted for.

To protect against this problem, we can, as just mentioned, make sure to update pointers appropriately when doing multiple compactions in a single cracker select, or, as we have chosen to do in our implementation, do the compactions always from tail to head, such that there is no potential for interference. Two values are inserted into the cracker index after a selection. The value being selected, and the minimally greater value.

Finally, we return the results to the query. If any compactions took place, we follow the edge pointers from the cracking scan to the offset array and then into the base index, giving us the relevant indices of the base columns.

4.3.2 Recognition

After recognizing a section of column which can be compressed, we need it store this information somewhere in order to later exploit it.

If we are doing per-fragment compression, then we can recognize a compression opportunity when two values are stored in the cracker index which are minimally different. In this case, we need no extra storage - the cracker index contains all the information we need in order to be able to later exploit this.

Otherwise, if we are applying a run-length encoding across the column, we need a separate structure to hold the information about each of the runs, since the compressions are in this case applied at a finer level of granularity than the cracker index is fit to store information about.

The main advantage of recognition versus compaction is that it avoids the various performance costs associated with applying compaction. Additionally, we have found that recognition methods are easier to implement than compaction methods.

To demonstrate an example of how per-fragment recognition compression works, look back to figure 4.3. In it, two uniform column fragments are shown at the end of a cracking scan, along with the two edge pointers and the current, unupdated cracker index. It is obvious that the selected column-fragment can be compressed, but the later column-fragment can now also be safely compressed for the same reason. We know in the cracker column the index before which all values are less and the index after which all values are greater, enabling us to exploit this uniform section in the future.

4.4 Run-length encoding

Our aim with run-length encoded (RLE) cracking is to be able to speed up the scan by enabling scanning pointers to hop over long runs of the same value. For this, we build and maintain information about all runs of consecutive nodes in the cracker column. This is done inside an array called *run_lengths*. *run_lengths*[*i*] indicates to a scanning pointer that the next `i` values (inclusive) are the same, and therefore can be considered together, whether that means that they are hopped over, or swapped away to another part of the column. Crucially, this applies when encountering *run_lengths*[*i*] in either direction, however, the information about in which direction is not stored, it is assumed that a pointer always visits the "start" of the run in the direction it is traveling. This is an invariant of our implementation.

In compactive RLE, the bi-directionality of the run-lengths stored is trivial, because the duplicated values are all deleted anyway, and only materialize upon tuple reconstruction. However, the complications associated with compaction still apply.

For recognition RLE, the bi-directionality is not a given however, duplicated values are not compacted, meaning that the existence of a run must be marked on both ends, such that a forwards moving edge pointer arrives at the front-side of the run and a backwards moving edge pointer arrives at the tail-side of the run. Figure 4.5 illustrates why duplicating the run-length on both sides of the run is necessary.

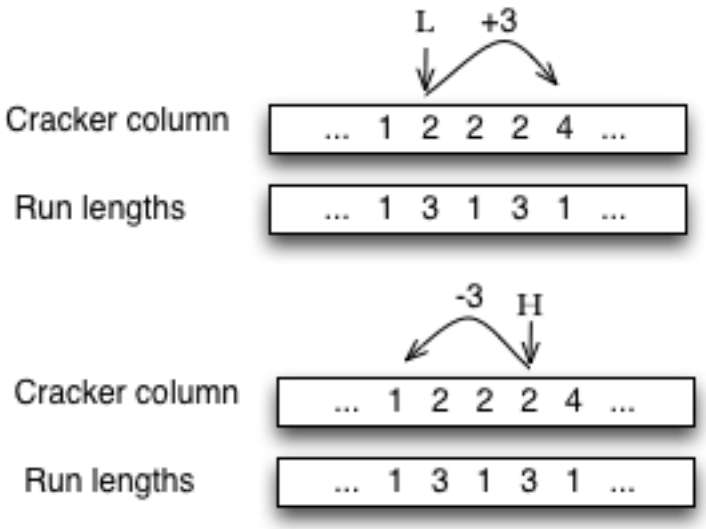

Figure 4.5: Both forwards and backwards moving pointers need knowledge about runs in order to exploit them

Note that the values of *run_lengths* within runs is not important, because in a correct implementation we will never read any of those values. In diagrams we will arbitrarily use 1s, although for our implementation any number in the range $[1, rl)$ can occur in *run_lengths* within a run of length rl.

We also need to maintain the *run_lengths* array under the restructuring that takes place during cracking to retain its consistency. We must ensure that the invariants associated with the edge pointers are maintained, and that the properties of *run_lengths* are preserved.

This must also take account of the fact that two runs of different length may need to be swapped during the scan. When making these swaps of different length runs, we may need to modify entries of *run_lengths* in order to correctly retain its properties. We have studied two approaches to swapping around runs of different lengths during the scanning phase.

The first approach to swapping two runs of different length is to swap the entire of the shorter run and modify entries of *run_lengths* corresponding to the longer run to maintain consistency after the swap. This means that the longer run is not fully swapped, so we have called it "underswapping". Underswapping causes the longer run to get broken up into two runs, one run the same size as the smaller run, which gets swapped entirely with the smaller run, and the other, a run consisting of the left-over section of the longer run, which is not swapped. Underswapping aims to

be simple to implement while maintaining the scanning benefits of run-length encoding, however runs are often broken and so large gains are not made.

The basic concept is illustrated in figure 4.6, in which the outlined green sections are swapped, the outlined red section stays still and the run length values outlined in blue have to be changed to preserve correctness of the *run_lengths* array after the swap.

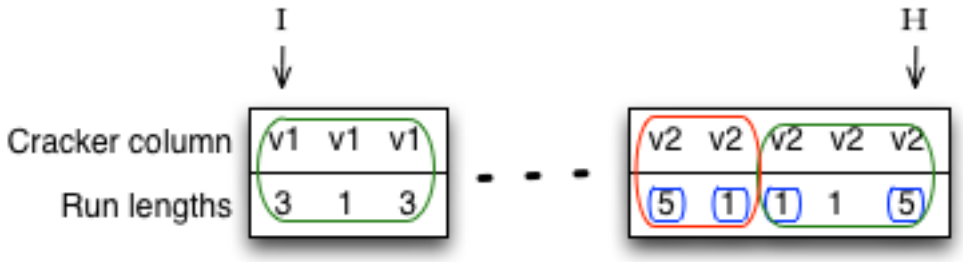

Figure 4.6: Example of Underswapping

The second approach is to swap the entire of the longer run, and pad the shorter run with more elements. This requires the padded elements to have their runs checked and potentially have entries modified, due to the possibility that they are constituents of another run. Due to the fact that the shorter run is swapped with more elements than are actually in that run (by padding), we call this "overswapping". Overswapping aims to preserve runs as much as possible, however it is also more complex. The diagram in figure 4.7 shows an example of using overswapping. The green outlined areas must be swapped, the red areas are also swapped with each other, and run lengths outlined in blue must be changed in order to retain run length correctness after the swap.

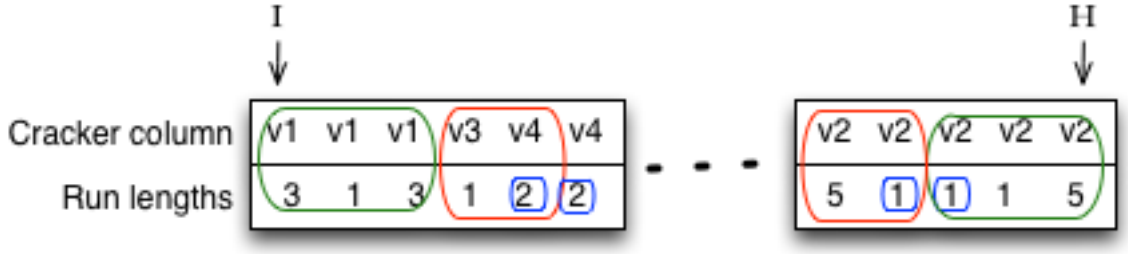

Figure 4.7: Example of Overswapping

Later in this section, we will discuss in detail the different cases for swapping around runs of different lengths, for both underswapping and overswapping. First we will look at the sections of algorithm that both of these RLE methods have in common.

Algorithm Overview

Both underswapping and overswapping implementations of cracking with RLE recognition compression share most of the algorithm in common. The only differences occur during the scan phase, however we will here describe the rest of the algorithm, which the two implementations share in common.

When the cracker column is initialized as a copy of the original column, *run_lengths* is also initialized. Its initial value is an array of the same length as the cracker column filled entirely with 1s.

The edge pointers are initialized from lookups into the cracker index and define the boundaries of the appropriate column-fragment, just like in other variants of the cracking algorithm.

The routines for tightening the pointers are different than for non-RLE variants, because the pointers move by the length of the run they are currently pointing to whenever they are tightened. Additionally, while tightening the edge pointers, we are looking for opportunities to build up and mark runs of duplicated values that they encounter.

When tightening the edge pointers in the tightening phase, we move pointers by jumping over each run, as well as building them up as we encounter them, while making sure to avoid under/overflow for cases where the value being sought isn't present. The way a run is built up during the tightening phase, using the low side pointer as an example, is shown in figure 4.8.

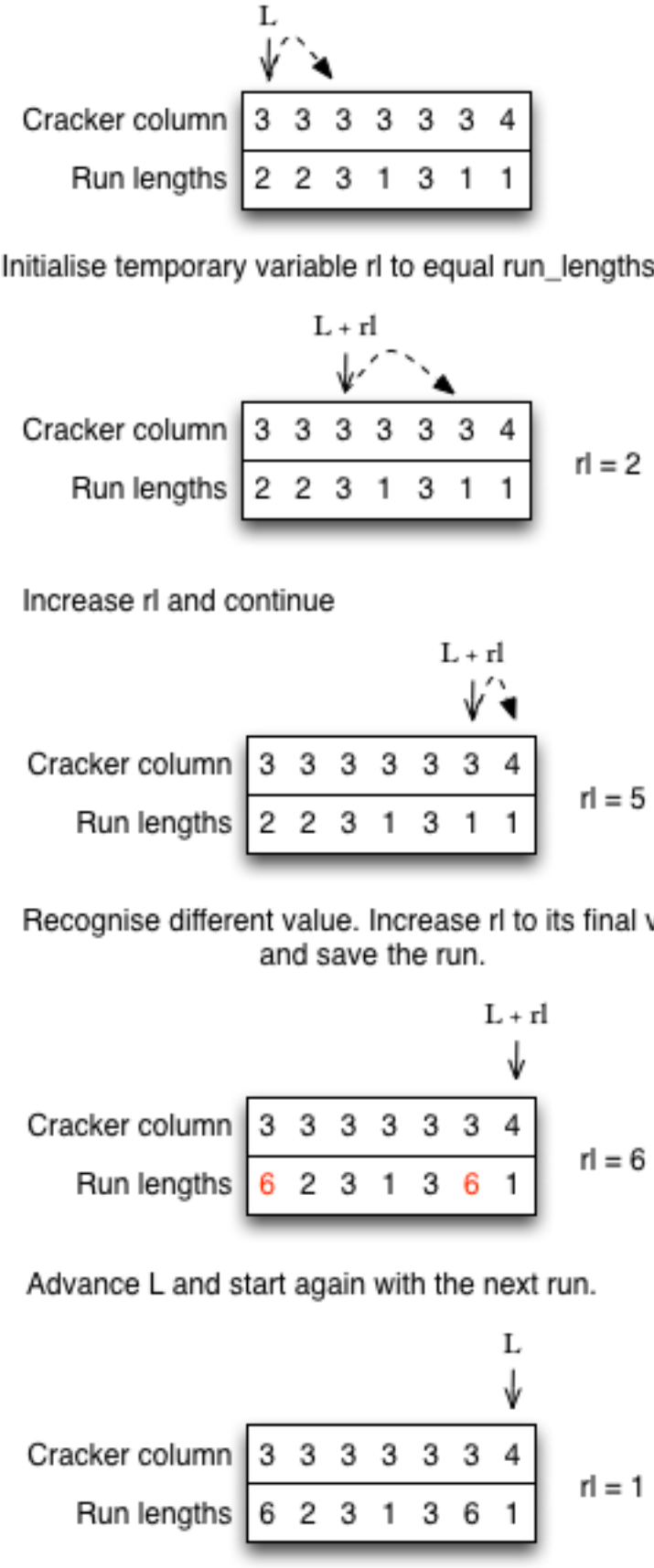

Figure 4.8: Run building with the low edge pointer

The case of swapping runs of different lengths between the low and iteration pointers is the

same for both underswapping and overswapping, because we have the knowledge that the values at the indices in the range $[I, L)$ are all the value being queried for. This means that we can immediately make a run out of it and do as big a swap as possible, advancing the pointers as much as possible. We did experiment with using the strict underswapping and overswapping methods for these stages, however we found that the exploiting the information we knew about the intermediate values between I and L gave better performance than either of them. Later in this subsection we describe the low-side swaps in more detail.

When the iteration pointer is to be advanced during the scan, we can choose to try to build runs during the advancement, or not. After trialling both methods on our generated datasets, we found that trying to build runs causes the algorithm to slow down, so our final implementations don't do that.

Also during the scan, the tightening of pointers after making swaps is the same as for standard cracking, including the fact that if the low edge pointer overtakes the iteration pointer during tightening, the iteration pointer catches up immediately.

After the scan, we check that values were indeed selected - if the low edge pointer has surpassed the high edge pointer, then we know from their relative invariants that the sought value isn't present in the column. Figure 4.9 illustrates this fact.

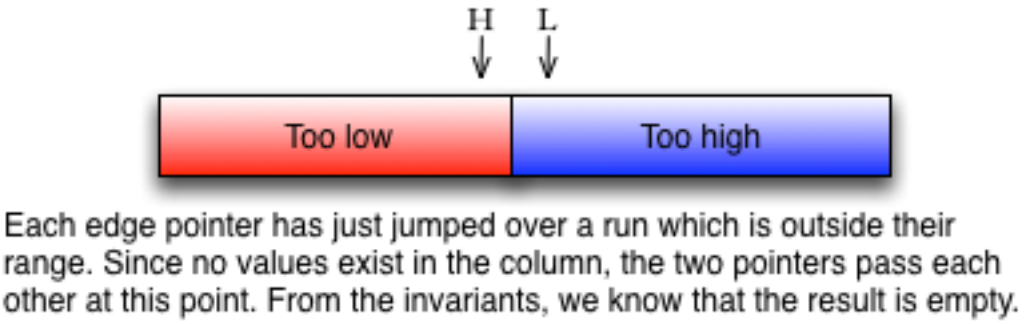

Figure 4.9: Two tessellating runs, neither of which are selected, form the boundary at which the edge pointers pass each other in a query returning no results.

Once the scan is finished, we combine into a single run the entire resultant fragment containing all instances of the value to be returned. We know this is the case for the same reason as when we do per-fragment compression. The insertion into the cracker index and returning of values is the same as for standard cracking.

Low side swaps

If the two runs to be swapped are the same length, then we can swap them immediately and entirely without doing anything else, because no runs are going to be invalidated and so none need to be edited.

If however they are different lengths, then we first change the situation in order to eliminate the possibility of having to deal with the situation in which values used to pad the shorter run spill into the longer run, creating an overlap. The change we make uses our knowledge that all the values of the cracker column lying within the index range $[L, I)$ are the selected value. We combine all of these values into a single run, resulting in the two runs to be swapped having a border just before the iteration pointer.

Our aim is to make the swap while preserving as much of the longer run as possible. To do this, we swap the entirety of the smaller run and edit the *run_lengths* array so that consistency is maintained.

Low side swaps: Iteration run shorter

Figure 4.10 shows the situation. The swap we intend to make is outlined in green and the run lengths to be amended are outlined in blue. We must ensure that after this swap takes place the properties of *run_lengths* are preserved.

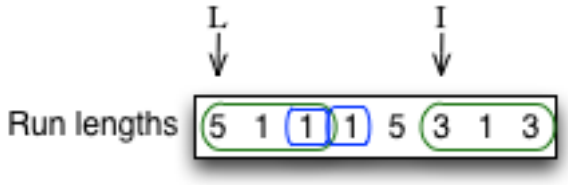

Figure 4.10: RLE low side swap, iteration run shorter: Swap recognized

Since we are swapping the full iteration pointer side run, we know that the last element getting swapped of those will be ending up at the tail end of our final run of selected values. Therefore we amend this entry in *run_lengths* with the number of values, which is equal to the difference between the two pointers, since all the values between them will constitute this run. Similarly, we know the element which will be at the start of the final run immediately follows the last element being swapped with the iteration run, so we amend that entry in *run_lengths* with the difference between the pointers as well. These amendments to *run_lengths* are shown in figure 4.11.

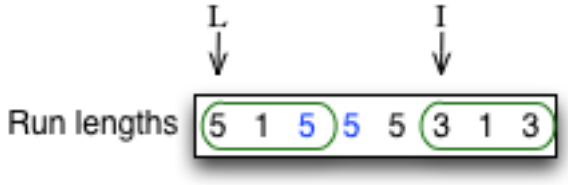

Figure 4.11: RLE low side swap, iteration run shorter: Run lengths entries amended

Having amended *run_lengths* appropriately, we can perform the full swap of the iteration side run with the appropriate number of elements from the low edge pointer, knowing that the resulting entries in the cracker column and *run_lengths* array will be consistent. Figure 4.12 shows the run lengths array after the swaps are completed.

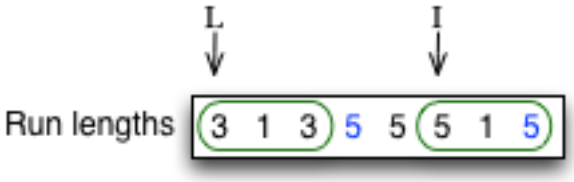

Figure 4.12: RLE low side swap, iteration run shorter: Swaps completed

At this point, we must advance the pointers to make progress in the scan. We know that the run we just swapped down to the low edge pointer contains values less than the selected node id, so the low edge pointer should be advanced beyond them. Also, the iteration pointer is now in the middle of a run, which is dangerous, so we get it out of there by advancing it to the end of that run. We don't swap it to the beginning because we know that it is a run containing the node id under selection - no need to scan it. The diagram in figure 4.13 demonstrates the advancing of the pointers.

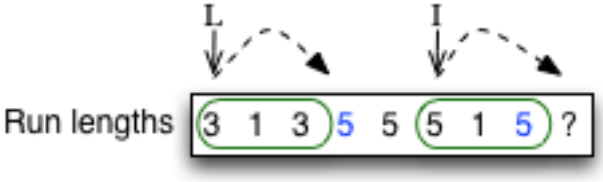

Figure 4.13: RLE low side swap, iteration run shorter: Pointers tightened

Low side swaps: Iteration run longer

In this situation, the shorter low-side run must be swapped to the end of the iteration run. We must again edit *run_lengths* to maintain consistency, as we can be seen from figure 4.14.

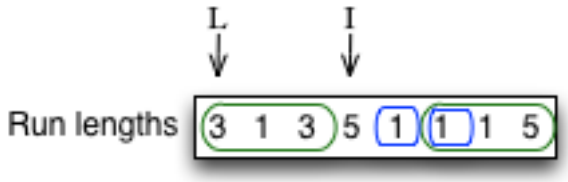

Figure 4.14: RLE low side swap, iteration run shorter: Swap recognized

The first element of the iteration run swapped with the low-side run must have its run-length changed, as well as the last element of the iteration run that is not swapped, so that after the swap, both sides of the run are still marked. The arithmetic for finding these positions is straightforward. Figure 4.15 shows the same situation as above but with the necessary amendments.

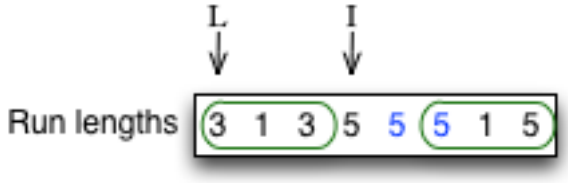

Figure 4.15: RLE low side swap, iteration run longer: Run lengths entries amended

After the amendments, the entire low-side run is swapped to the end of the iteration run, and the updated run markings mean that the iteration run is again wholly preserved, which you can see in the figure 4.16 .

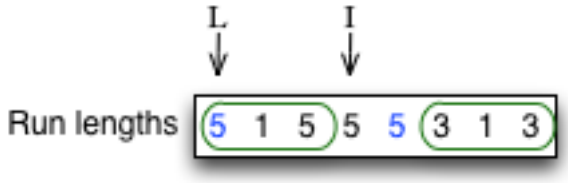

Figure 4.16: RLE low side swap, iteration run longer: Swaps completed

Finally, the pointers must be updated. The low pointer can hop over the swapped back run, since we know that its duplicated value is less than the selected value. The iteration pointer moves to the start of the run that was swapped forwards, the position of which was calculated earlier. Figure 4.17 shows how the pointers are updated.

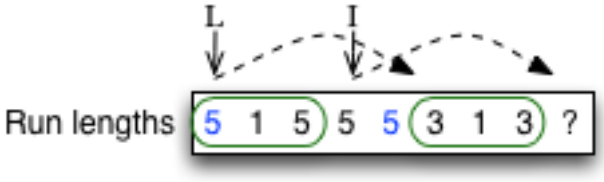

Figure 4.17: RLE low side swap, iteration run longer: Pointers advanced

4.4.1 Underswapping Scan

Underswapping is the simpler of the two approaches, because it has fewer edge cases to consider. We have only to amend the two pieces of the broken longer run before swapping all the values from the smaller run across to the equally sized run just created from a piece of the longer run. This method wastes some information we have acquired, but is fairly simple.

Within the cracking scan, if the run at the iteration pointer is to be swapped, it will either be swapped with the run at the low edge pointer or the high edge pointer. If the runs have equal length, they can be swapped immediately with no further amendments. Otherwise, the run which is longer can either be the run getting swapped towards the middle or towards the edge.

Under the following headers, we describe and illustrate the algorithm's behavior given the circumstance of the run at the iteration pointer. The two cases are all fairly similar and straightforward. We have lay them out in series of diagrams, starting at the initial situation, and ending with the runs swapped and the pointers appropriately tightened for the next iteration.

High side swaps: Iteration run shorter

The diagram in figure 4.18 shows the situation once we have identified that we are doing a high-side swap and the iteration run is shorter. The sections to be swapped are outlined in green and the run lengths which need to be amended are outlined in blue. We do not know about the intermediate values between the two runs, if there are any, so we have included an ellipsis in the depiction of the column to indicate these values.

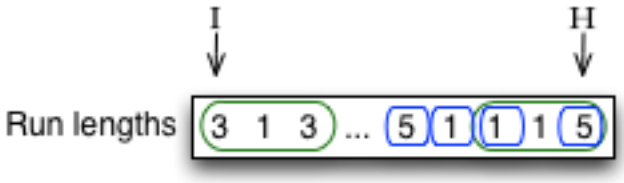

Figure 4.18: Underswapping high side run, iteration run shorter: Swap recognized

Figure 4.19 shows the amendments, which are shown with blue font. Due to underswapping, we can see that the long run of 5 has been broken into a run of 2 and a run of 3 in order to facilitate the swap. Below the run lengths array in the diagram is the cracker column, which we have filled with two generic values - i for the duplicated value in the iteration run, and h, the duplicated value in the high-side run.

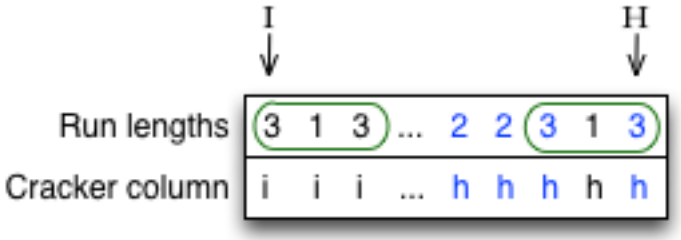

Figure 4.19: Underswapping high side run, iteration run shorter: Run lengths entries amended

After the swap, the values are moved and the correctness of run lengths and the cracker column is maintained from the amendments. We can also see that the run of i, known to be too great to be included in the selection, has been moved to the high-pointer and is ready to be hopped over.

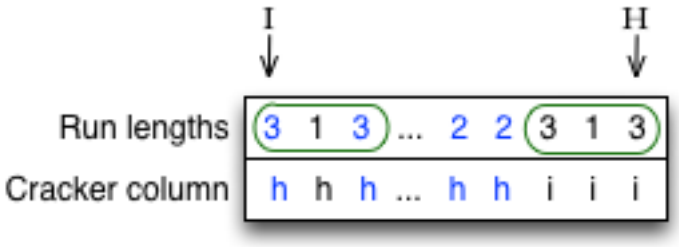

Figure 4.20: Underswapping high side run, iteration run shorter: Swaps completed

Finally, in figure 4.21 we show the tightening of the high pointer in which it hops over the swapped-up run.

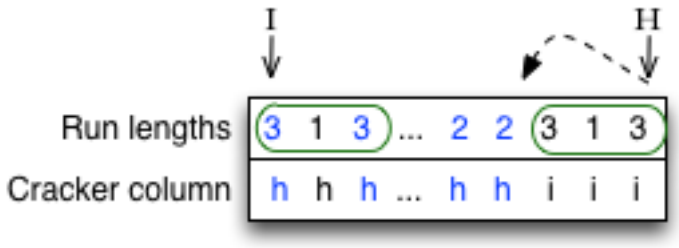

Figure 4.21: Underswapping high side run, iteration run shorter: High pointer tightened

High side swaps: Iteration run longer

As before, the sections to be swapped are outlined in green and the run lengths which need to be amended are outlined in blue. The ellipsis is also used in the same way as it was under the previous header. The situation is shown in figure 4.22.

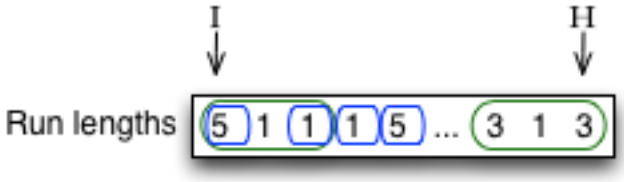

Figure 4.22: Underswapping high side run, iteration run longer: Swap recognized

Figure 4.23 shows the arrays after amending the necessary run lengths.

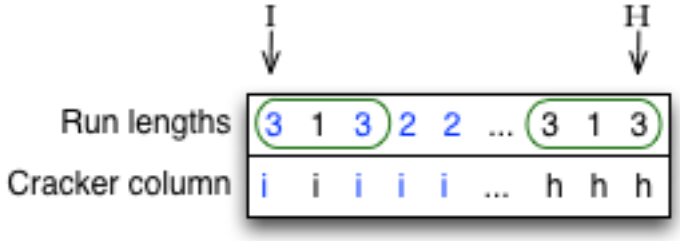

Figure 4.23: Underswapping high side run, iteration run longer: Run lengths amended

The amendments mean that after the swaps the runs are still all correct, as shown in figure 4.24.

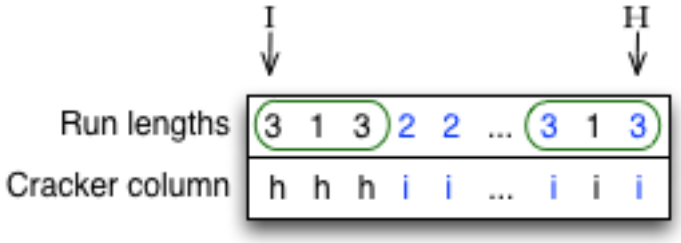

Figure 4.24: Underswapping high side run, iteration run longer: Swaps completed

Finally, the high edge pointer hops over the swapped up run to tighten the region to scan. This is shown in the diagram in figure 4.25.

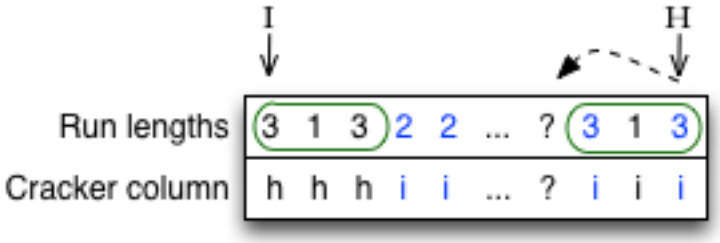

Figure 4.25: Underswapping high side run, iteration run longer: High pointer tightened

4.4.2 Overswapping Scan

When overswapping, we swap the entire of the longer run, and pad the shorter run to length. The runs within the padding must be amended to remain consistent, and if the padding contains partial runs, then those runs must be amended outside the padding as well to remain consistent after the swap. Additionally, if the padding causes the two swapped regions to overlap, then we don't have to swap all of the values - a saving that requires us to also make updates into *run_lengths* to preserve correctness.

Unlike in the low-side swap case, we know nothing about the values between the iteration pointer and high pointer, other than that the value at the high pointer is not greater than x.

High side swaps

As with low side swaps, when the runs are the same length, we can immediately swap them without any fuss and advance the edge pointer appropriately.

For high side swaps of unequal length, we have no information about the intermediate values between the two runs being swapped, therefore we have to manage the padding and ensure that no inconsistencies are introduced into *run_lengths* as a result. This includes the possibility that the padding may spill into the longer run and the possibility that the padding may form part of another separate run, causing that run to potentially become broken apart. We describe the first of these possibilities as "overlapping" because the padded shorter run overlaps with the longer run. The second we describe as the padding having a remainder, requiring amendments to *run_lengths*.

In our description of high-side overswapping, we will first describe the case in which the padding overlaps with the longer run. Then to describe non-overlapping padding, we will describe the strategy we used to deal with padding which has a remainder, followed by an explanation of the swaps which take place after the padding has been prepared.

Overlapping

Figure 4.26 is an illustration of the two possible cases of overlapping when swapping high-side runs. The values in the longer run which are also part of the padding are shown in red font.

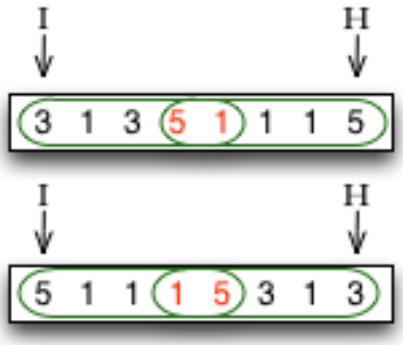

Figure 4.26: Padding can overlap with the longer run.

This problem is just like the low-side swap cases, but with slightly different arithmetic because H points to the tail end of its run. Because we assume that the *run_lengths* array is consistent, we know that we can take padding right up to the border with the other run and know that there is no possibility that the padding has any remainder. From this position we swap all the values between the iteration and high pointers except for the overlapping region, after making some adjustments in the longer run.

After the amendments are made we swap the two sections, the longer run aligns so consistency is preserved and the longer run is fully preserved.

Padding Remainder

The problem here is that the padding may contain for example two out of three elements of a run, meaning that if the padding gets swapped, the run will be broken. The padding may contain multiple runs - we need only check the last one. The ideal case is for the last run in the padding to be entirely contained within it, that is, the padding has no remainder.

Our approach to fixing padding remainder is to use underswapping. We break the padded run into a run inside the padding and a run outside the padding.

The implementation of this method is that we first initialize a padding pointer, with a view to move it through the padding until finding the last run. We then check if the padding has a remainder or not. If it doesn't, we don't need to do anything, otherwise, we fix the remainder into two runs as described in the previous paragraph.

Although the padding extends in different directions depending on which side it's on, the only thing that changes is the necessary arithmetic to fix the remainder if there is one, and to calculate the location of the swaps. We illustrate in figure 4.27 the amendments that must be made due to the padding remainder both in the case where the iteration run is longer and in the case where the high run is longer.

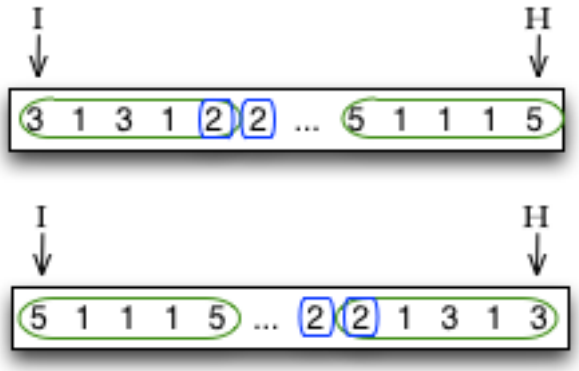

Figure 4.27: Padding can contain partial runs which must be amended.

Padding Swaps

At the stage at which we start to consider doing swaps, we have two equal length sections of column. The longer run is one of these sections, it is uniform. The other contains a run whose value we know and want to swap to the far side of the longer run, plus some padding to bring it up to length. The padding contains only whole runs and no runs will be broken by swapping it away, either by coincidence or because we have explicitly amended it.

We consider both of the two sections in two different parts. The shorter run, and the part of the longer run which will be swapped with it, are both one part, which we call the main part, because it constitutes the reason why we are doing the swap in the first place. The padding of the shorter run, and the corresponding part of the longer run we call the padding part, or just "the padding". In both the case where the iteration run is longer and where it is shorter, the padding is towards the middle-ground between the iteration and high pointers, as you can see from figure 4.27. After the swap therefore, the relative order of the main and padding parts will be swapped, meaning that we will have to make amendments in the longer run in order to preserve it. The run length entries which must be amended are outlined in blue in figure 4.28.

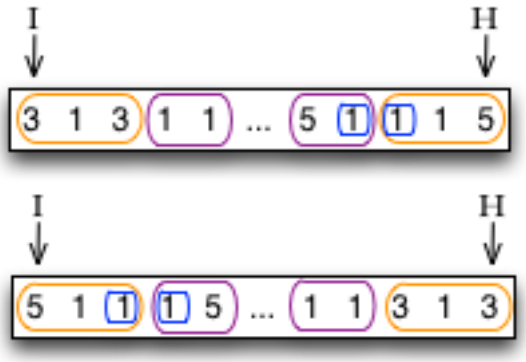

Figure 4.28: Amendments in the runs for swapping both the main and padding parts during a high-side overswap.

After the swap we are left with the iteration run swapped up and the *run_lengths* array in a correct state. Finally we tighten the high pointer by hopping it over the run that got swapped up. This is illustrated in figure 4.29.

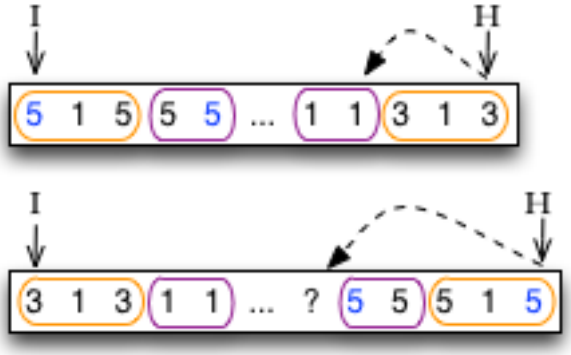

Figure 4.29: Tightening the high pointer after a completed high-side overswap.

Chapter 5

Evaluation

5.1 Experimental Setup

The experiments were performed on the department of computing lab machines, *sprite*30 and *sprite*29. These machines have a "HP EliteDesk 800 G2 TWR, Intel Core i7-6700 3.40GHz" CPU and have 16GB of RAM.

5.2 Datasets

5.2.1 LDBC Social Network Benchmark Data Generator

To test our work, we have used the LDBC-SNB DataGen program [2], which generates a social network graph to a number of scale factors. We used scale factors 1, 3 and 10. We would have used larger data but for the size of the machine we performed our experiments with. The number of nodes and edges in these graphs is shown in table 5.1.

Scale Factor	Nodes	Edges
1	9893	180624
3	24329	565248
10	65646	1938517

Table 5.1: LDBC-SNB generated datasets

A major use-case for adaptive indexing on graph data is the prospect of finding trends within a social network. When something happens in the news, for example, the workload changes, requiring a new index to improve performance. We have used a simple popularity/trend-analysis measure in pagerank ("personrank" in our case) to assess the performance of our system.

In this network, people are nodes and their relationships are edges. The node ids are unchanged from the data generation, they are not mapped onto consecutive integers from 0.

5.2.2 Randomly Generated Trees

We wished to find the break-even point of up-front sorting versus our implementations, so to support this, we used breadth-first search on randomly generated trees. To find the "break-even point", we find the time it takes to sort the edge array using quicksort [4] and then count the number of queries the cracking implementation has already answered in that time. This is used to determine the gains we can make by using adaptive indexing versus using offline index creation, that is, up-front sorting.

5.3 Graph Algorithms

5.3.1 Breadth First Search

Breadth first search, or BFS, involves choosing a starting node as the sole member of a frontier, which then expands in iterations in which members of the frontier append their out-neighbours which have not yet been visited to the frontier, and remove themselves. This continues until all vertices have been visited.

The most important factor for us in considering BFS is that it queries the outgoing edges of each node just once, meaning that compressed column fragments are never revisited during the course of the algorithm. For this reason we believe it provides an appropriate lower bound on the break-even point.

5.3.2 Pagerank

Pagerank [7] is a famous algorithm which stores a rank for each vertex and iteratively updates the values across the entire graph. All nodes are initialized with a rank of $\frac{1}{|V|}$. During each iteration, each node inherits from all of its in-neighbours a contribution of their pagerank divided by their out-degree. This value is then multiplied by a damping factor and then added to a base value to give the updated rank. The base value is defined as $\frac{1-d}{|V|}$.

In pagerank, every iteration considers all of the vertices and edges in the graph, and so over an execution of many iterations, any clustering or compression will see further benefits compared to BFS.

To benchmark our contributions, we have implemented pagerank as an equivalent "personrank" for the generated social network.

5.4 Systems under Comparison

The purpose of this project is not to create the fastest system, but to assess the applicability towards graph processing of compression-based variants of cracking. To ensure that our experiments fairly reflect this, we have chosen to compare our single-threaded cracking implementation to other single-threaded solutions. Predicated and vectorized implementations of cracking have been done and been shown to be highly effective for improving the CPU efficiency of cracking against the original algorithm [8], however, we have not applied these improvements to our algorithm. This is not a problem because our work is concerned primarily with the fundamental differences between standard cracking and cracking with adaptive compression, rather than optimizing them especially, although we touch on potential optimizations in our discussion of future work.

With that in mind, we have tested our system against the standard cracking algorithm and against upfront quicksort. The reason we have chosen quicksort specifically is for its similarity to cracking, in that it uses pivots to partially sort column sections. Compare this to cracking, which uses pivots to partition the column towards being fully sorted.

Our contributions featured variations in two dimensions: Opportunities to compress and storage of compression information. We have implemented per-fragment compaction, per-fragment recognition, underswapping RLE recognition and overswapping RLE recognition. The reason we decided not to implement compaction with RLE compression is that it seemed clear from the disappointing results of per-fragment compaction that it would not be worth it.

5.5 Results

5.5.1 Personrank

Using data generated by the LDBC-SNB at scale factors of 1, 3 and 10, we measured the wall-clock time to completion of pagerank implementations using each adaptive indexing technique. We have also included the results for an implementation using upfront quicksort.

For each of the three graphs, we ran pagerank over 50 and 100 iterations for every method. Each result is averaged over 10 runs. Blue and red represent the results over 50 and 100 iterations respectively.

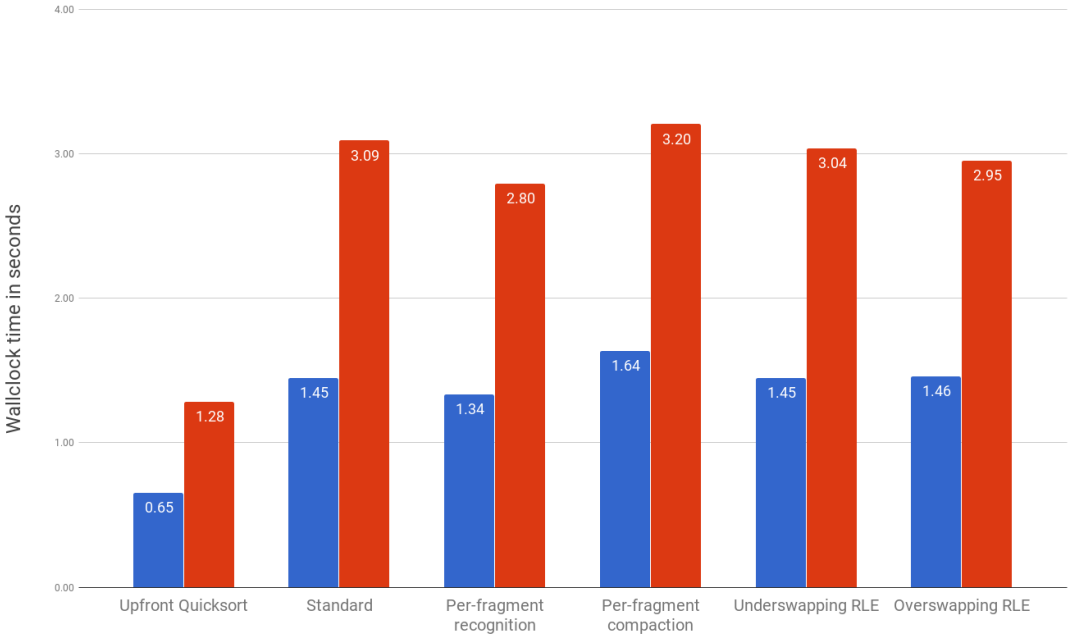

Figure 5.1: Personrank execution times for the SF1 social network graph

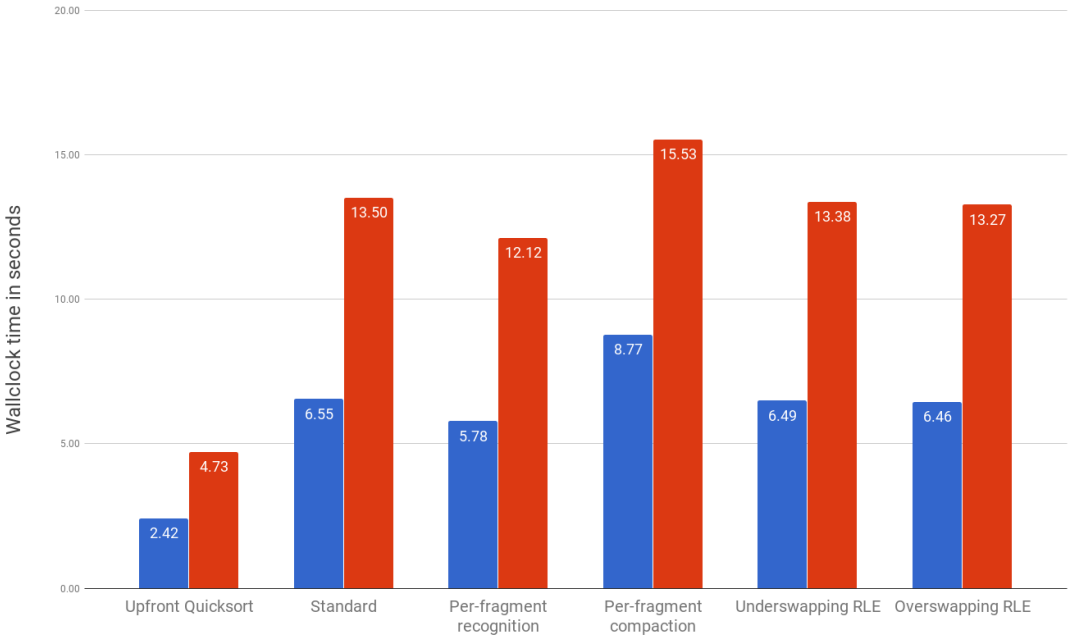

Figure 5.2: Personrank execution times for the SF3 social network graph

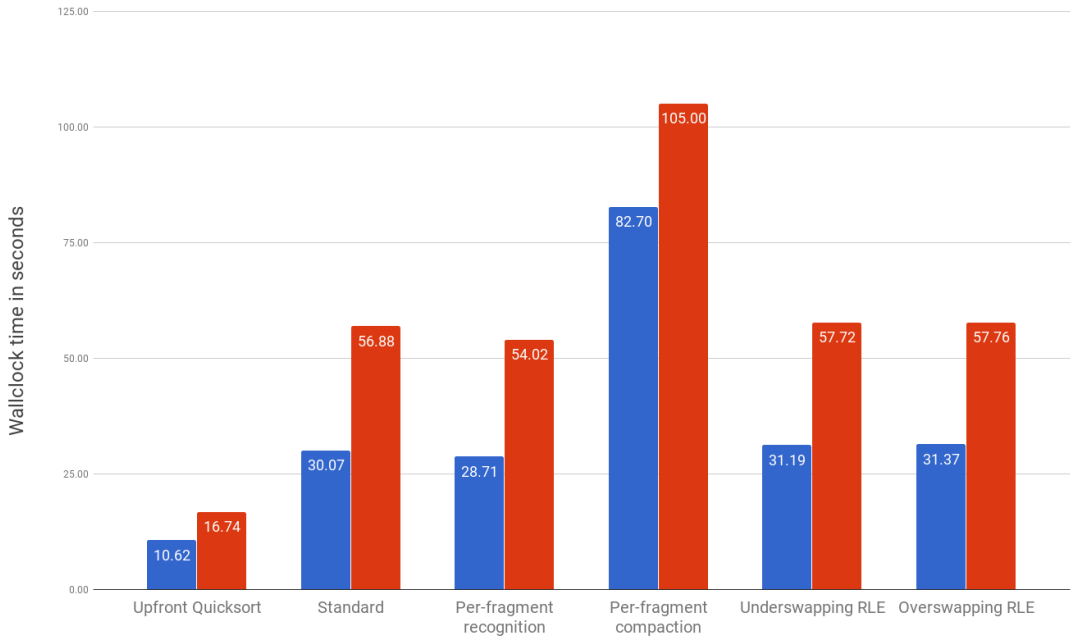

Figure 5.3: Personrank execution times for the SF10 social network graph

Our experiments showed that our implementation of compaction is always the slowest, regardless of data size or the number of iterations. This is due to the prohibitively high costs associated with the constant copying of memory which this technique depends on.

We also showed however, that per-fragment recognition was always the fastest. This is because, by recognizing a uniform column-fragment, we can avoid doing any scanning at all. This optimization to the original algorithm yielded a small average speedup across our six reported executions of 8.1%.

The two RLE methods we developed showed promise. Across our reported executions, they were faster than standard cracking as often as they were slower. We found very little difference in performance between underswapping and overswapping, although this is likely to be because they are inherently very similar, differing only in the way they perform high-side swaps of two runs of different lengths - a case which is evidently not encountered enough to show any sizable performance difference between the two different implementations of that case.

5.5.2 Break-even Point

Using randomly generated trees, we ran BFS using each adaptive method and counted how many cracking queries were completed in the same amount of time as it took to sort the tree using quicksort.

The reason we chose to do BFS on trees, is that it represents a case in which every query is guaranteed to be doing a scan and not returning compressed values. We did this in order to assess how much of an impact on the break-even performance and early-scanning speed our changes had to the original cracking algorithm.

We evaluated the break-even point for trees of size 100,000, 500,000 and 1,000,000 (number of nodes), averaged over 10 runs, with a new tree for each run. Our results are shown in figure 5.4. Blue, red and yellow represent the results for the trees with 100,000, 500,000 and 1,000,000 nodes respectively.

Compaction performed poorly compared to standard cracking in this test. We see that with greater data size, the number of queries ran by compaction in the same time as it took to sort the array decreases. This makes sense given that the primary cost factor in compaction is the copying of memory. As the size of the array increases and the number of copied values increases, so the performance during early queries will decrease.

Per-fragment recognition makes no changes to the cracking scan, and so it makes sense that

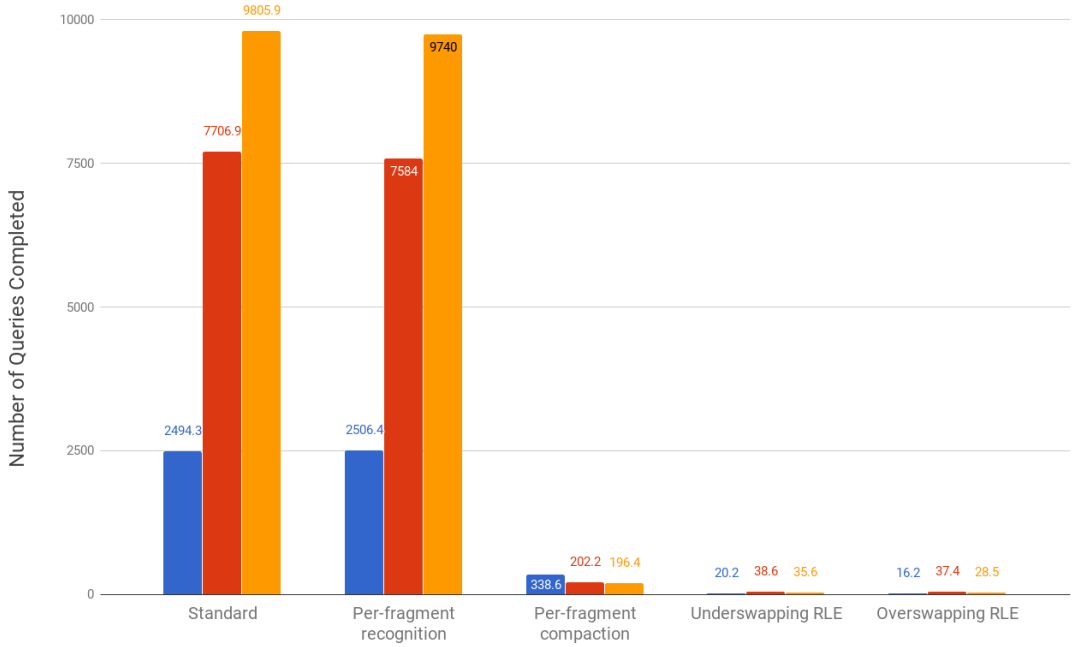

Figure 5.4: Break-even point for each studied adaptive technique

there would be little difference in the break-even point compared to standard cracking, regardless of the data size.

The two RLE cracking variants actually change the implementation of the scan - they perform extra work to build and maintain knowledge about runs in exchange for the ability to exploit this knowledge. In the early queries we expect the costs to outweigh the benefits, because at the start, there is no run-length information to exploit, but at the same time there is a lot of new information to build and write into memory, which is expensive. This results in a significantly higher relative cost for early queries when using RLE cracking, causing the break-even point to be dramatically lower. This is unfortunate, because one of the main advantages of using cracking over a non-adaptive indexing technique is that it is lightweight - a property lost by performing expensive run-length encoding in early queries.

Compared to overswapping, underswapping performs less work during high-side swapping. It has fewer branches because it does not have to deal with padding or overlapping. The earliest queries perform the largest scans and therefore perform the most swaps. It therefore makes sense that the relative advantage underswapping has in performing less work during swapping would be most evident in the earliest queries.

5.5.3 Summary

- **Per-fragment Compaction**: This method was quite disappointing, although it cannot be said that it was unexpected. The costs of deleting arbitrary elements from an array and copying the non-contiguous data towards the head are too high to make this a viable technique.
- **Per-fragment Recognition**: By recognizing uniform column pieces using the invariants of the cracker index, we can prevent unnecessary scans, which improves performance. We showed that per-fragment recognition is an effective way to improve the performance of cracking as an adaptive index on an edge-array representing a graph.
- **RLE**: We showed that run-length encoding an edge array during cracking is a technique that performs similarly to standard cracking in terms of overall speed, but is a far less lightweight operation.

Chapter 6

Conclusion

6.1 What we did

We implemented four compression-based variations of the original cracking algorithm and applied them to graph data. We evaluated our implementations using pagerank and by finding the break-even point point for each algorithm, comparing their performance to that of standard cracking and to upfront sorting.

From the evaluation of these methods, we found compaction to be unsuitable due to the prohibitively high cost of compacting values. We found run-length encoding to perform similarly to standard cracking for overall speed in the execution of pagerank, however we found that the costs added to the scanning phase cause early queries to not retain the lightweight property of standard cracking. This caused its break-even point against upfront sorting to be far lower. Finally, we found a positive result among our explored variations - recognition-based compression was shown to be an improvement over standard cracking for our pagerank experiments, with the highest recorded cost to the break-even point being less than 2%.

6.2 Future Work

- **CPU efficient implementations**: None of our implementations used predication or vectorization in order to improve their CPU efficiency, which is known to be effective[8]. Standard cracking sees a significant speed up in single-threaded performance for the predicated and vectorized implementation. It would be more complicated for the run-length encoding variant of cracking, because the implementation for standard cracking is fairly dependent on processing a single value at a time. This is the case for the backup and active slots for the predicated backup technique and also in accounting for buffer overflow in the vectorized implementation.

- **Parallel implementations**: We studied only single-threaded implementations of our algorithms. Parallel implementations of them present interesting difficulties, because the refined partition & merge algorithm[8] could not be naively copied over to a run-length encoding form of cracking - we would need a way to initialize the multiple edge pointers used such that they did not lie within runs. Furthermore, when swapping runs of different lengths, we would frequently encounter situations in which the high-side block of a separated block within the refined partition & merge could not hold the different sized run due to overlapping with another block. These difficulties might suggest that we would be better off using the simple partition & merge, despite its poorer performance, however that is all a possibility for future work.

- **Stochastic RLE Cracking**[3]: One of the properties of stochastic cracking that makes it an appealing choice for application towards RLE is that it improves the workload-robustness of it. In that, it aims to prevent unnecessary physical reorganization and scanning - a property that RLE cracking would benefit greatly from, given that the cost of scanning and swapping in RLE cracking is what makes it so much less lightweight that standard cracking.

- **Different compression schemes**: We used run-length encoding, but we didn't look at any other compression formats, such as compression formats optimized for different types of graph queries, such as point queries, which aim to find matching subgraphs within a database.

References

[1] G. Aluc. *Workload Matters: A Robust Approach to Physical RDF Database Design*. PhD thesis, University of Waterloo, Ontario, Canada, 2015.

[2] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 619–630, New York, NY, USA, 2015. ACM.

[3] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. *Proc. VLDB Endow.*, 5(6):502–513, Feb. 2012.

[4] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[5] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *Proceedings of the 3rd International Conference on Innovative Data Systems Research (CIDR)*, pages 68–78, Asilomar, California, 2007.

[6] F. McSherry, M. Isard, and D. G. Murray. Scalability! but at what COST? In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*, Kartause Ittingen, Switzerland, 2015. USENIX Association.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[8] H. Pirk, E. Petraki, S. Idreos, S. Manegold, and M. Kersten. Database cracking: Fancy scan, not poor man's sort! In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 4:1–4:8, New York, NY, USA, 2014. ACM.

[9] J. Shun and G. E. Blelloch. Ligra: a lightweight graph processing framework for shared memory. In *ACM Sigplan Notices*, volume 48, pages 135–146. ACM, 2013.

[10] Y. Zhang, V. Kiriansky, C. Mendis, M. Zaharia, and S. Amarasinghe. Optimizing cache performance for graph analytics. *arXiv preprint arXiv:1608.01362*, 2016.