Óbuda University

PhD Thesis

Data Compression and Data Management in Stream and Batch Processing Environment

István Finta

Supervisor:

Dr. habil. Sándor Szénási

Doctoral School of Applied Informatics and Applied Mathematics

Budapest, 2021

Final Examination Committee

The following served on the Examining Committee for this thesis.

Internal Opponent:	Dr. habil. Edit Laufer, PhD Óbuda University
External Opponent:	Dr. Ladislav Végh, PhD J. Selye University, Komárno, Slovakia
Chair:	Prof. Dr. Aurél Galántai, DSc, professor emeritus Óbuda University
Secretary:	Dr. Gábor Kertész, PhD Óbuda University
External Member:	Prof. Dr. Róbert Fullér, DSc University of Szeged
Internal Member:	Dr. habil. Imre Felde, PhD Óbuda University
Internal Member:	Dr. habil Márta Takács, PhD Óbuda University

Declaration

I, the undersigned, **István Finta**, hereby state and declare that this Ph.D. thesis represents my own work and is the result of my own original research. I only used the sources listed in the references. All parts taken from other works, either as word for word citation or rewritten keeping the original meaning, have been unambiguously marked, and reference to the source was included.

Nyilatkozat

Alulírott **Finta István** kijelentem, hogy ez a doktori értekezés a saját munkámat mutatja be, és a saját eredeti kutatásom eredménye. Csak a hivatkozásokban felsorolt forrásokat használtam fel, minden más munkából származó rész, a szó szerinti, vagy az eredeti jelentést megtartó átiratok egyértelműen jelölve, és forráshivatkozva lettek.

Kivonat

Az időről-időre megjelenő különféle enabler technológiák megkövetelik, hogy átgondoljuk és szükség esetén átalakítsuk az érintett felhasználási területen az addig elterjedt mérnöki gyakorlatot. Az informatika és a számítástudományok területén az egyik ilyen enabler technológia a Big data megjelenése volt: korábban soha nem látott mennyiségű adat tárolása és feldolgozása vált lehetővé megfizethető áron és ésszerű(reasonable) időben. Természetesen ebben az esetben is az általános technológiát/módszert testre kell szabni az alkalmazó igényeihez és lehetőségeihez mérten. A Big data megjelenése közvetlenül az elosztott rendszerű/szemléletű adatmenedzsmentre és az állományszervezésre (közvetve pedig az adatokból információt kinyerő analitikai, AI/ML technológiákra) gyakorolta a legnagyobb hatást.

A technológia a hatalmas adatmennyiségek ésszerű időn belüli kezelésén keresztül lehetővé teszi, hogy igénytől és területtől függően nagyobb arányban kapjunk valós idejű képet egy adott eseményről vagy megfigyelt rendszerről. A felhalmozott historikus adatokra épített analitikai megoldásokkal pedig pontosabb előrejelzéseket (predikciókat) készítsünk. Azonban ehhez, a minél pontosabb előrejelzések érdekében, tiszta adatokra van szükségünk. Másfelől, éppen az adatok hatalmas mennyisége miatt, sok esetben célszerű az adatokat tömörítve tárolni és/vagy mozgatni.

A disszertációm elején rövidem bemutatom/jellemzem a további vizsgálataim alapjául szolgáló távközlési környezetet, amiben a hagyományos adatfeldolgozást egy proof-of-concept keretében Big data alapúval helyettesítettem. Egyúttal kijelölöm azt a két kutatási területet is, a veszteségmentes tömörítés és a duplikáció kezelése, aminek az eredményeit ebben a disszertációban foglaltam össze.

Az első kutatási területhez/téziscsoporthoz kapcsolódóan bemutatom az általam kidolgozott veszteségmentes tömörítési algoritmust, ahol a korábbi módszerekhez képest számítási erőforrásra cserélem a tárolási erőforrást. Bebizonyítom, hogy az algoritmus helyesen működik. Az elvégzett elemzések alapján bemutatom a legjobb és legrosszabb eseteket a tömörítési arány, a feldolgozási idő és a felhasznált tárhely tekintetében. Ezeket az eredményeket összehasonlítom az általam kidolgozott módszer alapjául szolgáló algoritmussal. Rámutatok a legrosszabb eset bemeneti mintázat meghatározásának nehézségére, amivel kapcsolatban megfogalmazok egy szükséges feltételt.

A második téziscsoport során bevezetek egy stream processing környezetbe szánt, sűrű kulcstérben hatékonyan működő, duplikáció szűrő adatszerkezetet, az IMBT-t. A továbbiakban belátom az IMBT-ről, hogy helyesen működik. Bebizonyítom, hogy az adatszerkezet teljesítménye a kulcsok száma mellett azok statisztikai eloszlásától is függ. Kezdetben speciális kulcs-eloszlásokra, eloszlás osztályokra, vonatkoztatva vezetek le zárt képleteket a keresési költséggel kapcsolatban. Majd számszerűsítem, hogy az IMBT milyen feltételek mellett mekkora előnyt mutat más adatszerkezetekkel szemben. A mátrixos ábrázolás során pedig olyan számítási eszközt mutatok be, amivel tetszőleges kulcseloszlás modellezhető, így szimulácók segítségével közelítő képletek adhatók az IMBT hatékonyságával kapcsolatban. Végül bemutatom az IMBT első verziós elosztott környezetű működését.

Abstract

The enabler technologies appear time-to-time and force us to re-think or even reshape the status quo or the best practices applied in that particular area so far. One such kind of enabler technology in the field of computer science and engineering was the Big data: it made possible to store and process such a huge amount of data for affordable price and within reasonable period of time like never before.

Obviously, the universal technologies and methods requires some sort/extent customization based on the needs and the possibilities of the application field. The appearance of 'Big data' had direct influence to the field of distributed systems, including the scope of data management and data organization, and indirectly to field of data science, which by now may extract the information from much larger data-sets, than ever before.

The new technology makes it easier to get near realtime insight into an observed system and/or to create more accurate predictions through the higher amount of accumulated historical data, based on the need of the given application area. However, for higher accuracy the clean data is essential. Additionally, due to the enormous amount of raw data, it is reasonable to apply some sort of compression method during the storage and/or transmission of the data.

In the course of the introduction of this dissertation I briefly characterize the telecommunication environment, in which the traditional data processing and pipeline had to be replaced with Big data based technologies as a proof-of-concept. At the same time I delineate those two research areas, the lossless data compression and duplication handling, which are in the scope of this dissertation.

In the first theses-group I introduce a lossless data compression algorithm, where the memory resources had been replaced by computation resources. I prove that the algorithm works correctly. Based on the analyses I reveal the best and worst cases in terms of compression ratio, processing time and memory need. I compare these results with the initial algorithm, from which my idea was derived. Then I point out to the difficulty of the determination of the worst case input pattern, relating to which I determine a required condition.

In the second thesis-group I introduce a data structure, the IMBT which is meant to be used as an efficient filter in a stream processing environment and performs efficiently in case of dense key-space. Then I prove that the IMBT works correctly. I point out that the performance of the data structure, next to the number of keys is a function of their distribution as well. Assuming special key distributions I introduce closed formulas which work correctly in the context of the given distributions. Then, based on the formulas I quantify the advantages/disadvantages of the IMBT, that is, I can formulate distribution dependent conditions. In order to be able to model arbitrary distribution in a computationally convenient way I introduce the matrix representation: through fast mass-simulations well fitting formulas can be gained. Finally I show the operations of the data structure in distributed environment.

Acknowledgements

I would like to thank to my manager, Lóránt Farkas for the trigger, and my supervisors Sándor Szénási and Szabolcs Sergyán for their continuous support during my way. Nokia Bell Labs and Óbuda University always provided the vibrating and inspiring environment and the challenging tasks as well.

Dedication

I dedicate this work to my family.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Consider an environment where the group of distributed measuring instruments, let's say k instances $(M_1, M_2, ...M_k)$, emit their measurement reports R_{Mi} (Fig. 1.1). Each instrument has own unique identity. My goal is to collect, normalize and transport of these measurement reports, along with guaranteed duplication filtering and high-speed (near real-time) processing.

Thermometers of the national weather service might be an example of endpoints for such a system.

Figure 1.1: Meters with emitted measurement reports

Regarding monitoring systems it is a natural expectation to get real-time insight about the investigated system. However, the size and complexity of the system under investigation, the sort of the collected data, the influence of the measurement type to the observed system and several other factors highly influence if which extent the above expectation can be fulfilled.

In info-communication networks two types of reports are generated about the events in the network nodes: periodic interval emitted (periodic timing based) synchronous and event driven asynchronous reports. The locally generated reports has to be transmitted into the permanent storage systems and has to be prepared for post-processing. It is essential to consider this kind of distribution during observation and data collection regarding networked systems.

Until the first half of the '10, in accordance with the regulatory, the dominant data pipeline based on batch processing and made available both the raw and several months aggregated data/KPIs with significant delay (Fig. 1.2). Due to technological and economical reasons the real-time available information, expressed in the percent of the totally

observed data, was marginal. That is, the fine-tuning of the parameters of a network mostly based on historical data. The accuracy of error prediction was also limited by the batch processing caused delay.

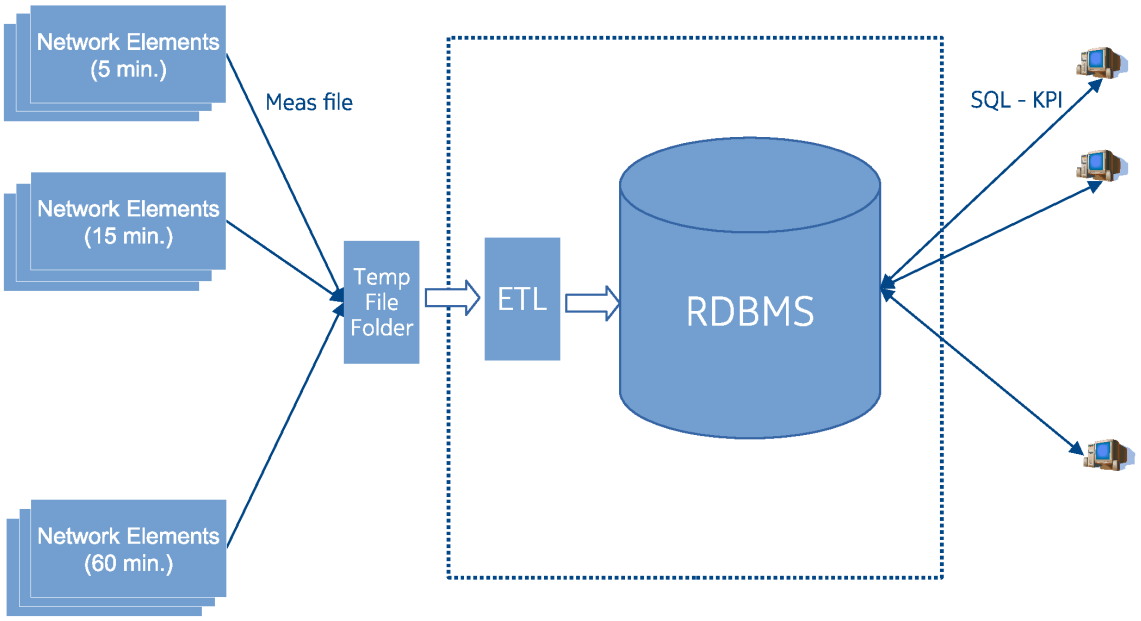

Figure 1.2: Traditional data pipeline in telco environment

The higher percentage of real-time data processing from the second half of '00 might happened due to the appearance of the enabler technologies, which is called Big data nowadays: by that time the research results from both the academia and the internet tech company sides made possible to store and process enormous amount of data. At that scale the Map-Reduce [2] (from Google, a batch processing programming paradigm), Google File System (-GFS) [3], the Hadoop Distributed File System [4] and MAPR [5] were among the pioneers. On the other hand the emerging cloud services (with the help of which the resources usage can theoretically always be kept close to the optimal) were also required faster and more accurate measurement data processing for proper functionality.

The rapidly spreading of the social media platforms implied that the backend needed to be able to manage in a very short period of time the suddenly appearing several ten- or hundred millions uploaded photos and posts. In the near real-time data processing technologies Twitter was one of the pioneers by the in-house developed STORM [6] stream processing framework at the beginning of '10. Storm utilized ZeroMQ [7], NETTY [8], RABBIT MQ [9] or KAFKA [10] as a lower level messaging services. At that time Google Millwheel [11], LinkedIn Samza [12] or UC Berkley Spark [13] also considered as determinative streaming frameworks.

For freshly founded companies implementing their relevant parts of the business model in Java, Clojure or Python to conform the above Big data technologies is not mean a problem. However, for most of the existing companies SQL is the de-facto standard in the field of business processes. In order to be able to fit the Big data systems, which are running on cheap hardware, to the existing SQL based/supported business processes the upward compatible middlewares appeared, like HBASE [14] or HIVE [15]. These middlewares are able to translate the SQL queries to MapReduce jobs, for instance. Other data base examples,

which operate in near real-time environments: Cassandra [16], CouchDB [17], VoltDB [18]. Additionally, the proper functioning of these distributed systems require such monitoring systems and rutins, which provide the continuous operation, fault-tolerance, etc., like ZOOKEEPER [19], GANGLIA [20].

In the meantime the LAMBDA architecture [21] concept also appeared, which is a mixture of the batch and stream processing paradigms. These architectures might not be the fastest ones, however, due to the presence of permanent storage (through the batch processing side) their resilience against data loss is quite high.

1.1 Efficient Filtering

Earlier I participated in an industrial research project, which aimed the investigation of the applicability of Big data technologies in telecommunication industry. The main drivers behind the project were the followings: first is to seek for alternatives compared to vendor locked traditional relational data bases, replacing them with theoretically cheaper, open-source NoSQL technologies, Fig. 1.3. Second, is to increase the real-time processing ratio of measurements data.

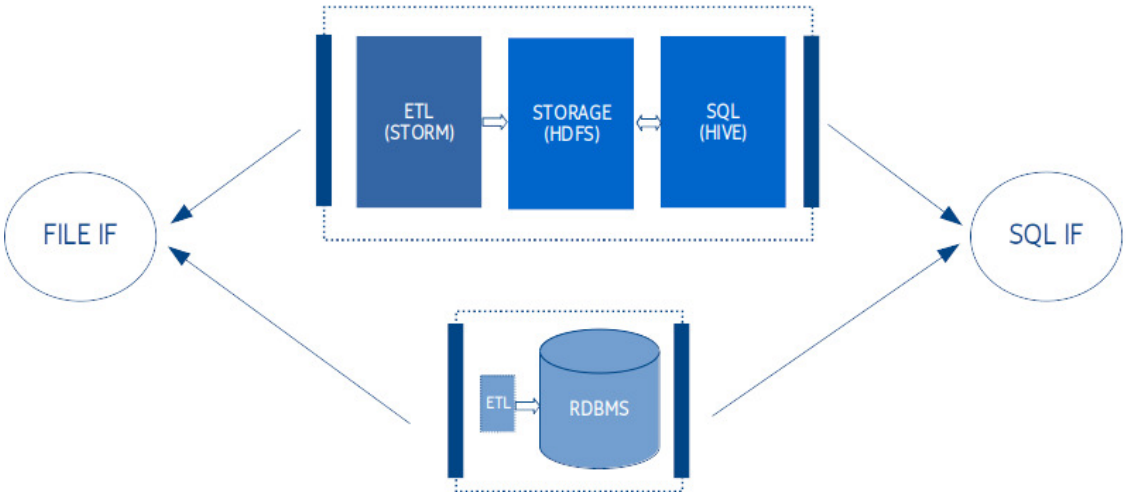

Figure 1.3: NoSQL replacement

During my job I had to prepare a kind of prototype data pipeline, which ensures the duplication free storage in the persistent file system (HDFS), independently if type of duplication is external or internal, origins due to some sort of re-transmission. Otherwise the duplication makes dirty the inherently clean data, and as a result distorts the derivative statistics or cause extra effort and cost during the late cleaning. In order to keep the raw data clean during the ETL process I have investigated several traditional or the previously mentioned in-memory DBs. However, none of them could meet with the performance expectation next to the external communication costs. Therefore I paid my attention towards built-in abstract data structures, like SET and MAP, provided by the Java Collections Framework (JCF) [22]. Behind these abstractions the models are mostly some sort Binary Search Tree [23], [24] (AVL-tree [25] [26] [27], RB-tree [28], etc.) or a hash table [24]. Since the prepared pipeline was modular by design, I could examine the

application of other data structures as well, like B-tree [29] [30], (a,b)-Tree [33], Interval-Tree [34] [35] [36]. Then I have examined the application of stand-alone layers for filtering purposes, like Bloomfilter [39] or Chord [61].

The problem with the above implementations were either the filtering was not accurate, or the space complexity was linearly proportional with keys processed so far. Therefore, always could be found such long operational period of time, when the oldest keys had to be removed from the filter so that let enough space for the new keys. With this strategy either the loading of older keys were limited,or let the presence of duplication in case of older keys.

From processing point of view the performance degradation appeared gradually in logarithmic manner, instead of a sharp decrease, along with the increasing number of keys.

The hash tables performed with the expected O(1) time complexity up to their preset capacity. Then the effect of 're-hashing' decreased the performance of the pipeline with such extent, that so called SWAP-ing took place and the pipeline collapsed, actually.

Therefore my goal was to work out such a (that kind of) filtering mechanism, which is far more memory-friendly than the earlier solutions, but still could be described/characterized with O(1) time-complexity. So, in this very special case I have worked out unique sequential numbering method which, based on a computation, always associates the identical sequence number to the identical data, independently from the entry point of the data. The replacement of mnemonic identities by unique sequence numbers made possible that in the stream processing framework, which acted as a high speed filter as well next to ETL purposes, only the efficient filtering of the sequence numbers had to be implemented.

1.2 Compression

Another scope of my research related to the efficient storage of the individually n-times KBytes measurement data, since both the one-by-one transmission and storage of the inherently small sized files could significantly reduced the efficiency of Big data file systems, where the block size is typically equal or greater than 64 MBytes(!). The previously described problem is called 'small file problem' [40], [41], [42] in the literature. To avoid the problem several methods had been worked out, mostly with embedding of small files into so called container files, like Sequence or Map files [40]. Since the daily generated new data exceeds the n*10 GBytes even in an average sized mobile network and, in accordance with the regulatory in Europe, this data have to be stored at least for two years, I had to examine the application of lossless data compression algorithms, like LZ77 [43], [44], [46], LZ78 [47], LZW [48] [51] and LZAP [44].

Access pattern is such an important parameter [40] which predicts both the frequency and the access type to the stored data on some basis. The measurement files are actually immutable ones, therefore the expected access pattern in this case is WRITE-ONCE–READ-MANY-TIMES. It is worth to consider this information during the selection and application of compression methods as well, since both the encoding and decoding time complexities may be highly optimized with the properly selected algorithms.

Access pattern driven compression is so seriously/strictly handled by Google that in their own invented compression method, 'brotli' [52], the coders work from a pre-defined,

pre-weighted, non-volatile static dictionary, which comprises 13K entries approximately. In web environment this approach significantly boosts the procedure. Prior to brotli, in 2011 Google introduced another compression method, which is soon become widespread in Big data technologies the so called Snappy compression [53].

Next to Google, also Facebook introduced their own compression method, Zstandard [54] [55], in 2016, which is a mule of LZSS and Huffman encoding [44]. Zstandard focuses on the decoding side performance, next to the best available compression ratio.

First, I reviewed the above methods and algorithms, then I came up with my customized relatively fast, easy re-weight, memory-friendly, LZW based virtual dictionary extension solution. This customized method is aimed to be asymptotically optimal next to show excellent compression ratio from solid compression point of view, where the data files (the inputs for solid compression) can be characterized with relatively many and relatively long recurring identical patterns at the beginning, like the measurement headers.

1.3 Goal of the Research

In case of efficient filtering the goal of my research was to work out the concept of the required data structure. Next to the initial theoretical examinations I have built the prototype as well and investigated the behavior of the IMBT under different circumstances, with the help of the simulation and experimental results. The experimental results lead to novel theoretical relations between incoming key distributions and the advantage of IMBT compared to other data structures.

In case of customized compression the goal of my research was to work out the detailed virtual dictionary extension method, both the encoding and the decoding side and determine the main theoretical relations. Parallel to the theoretical work I implemented the prototype and tested the theory through experimental results.

In the following first the Virtual Dictionary Extension, then the efficient filtering related research and outcomes will be presented.

Chapter 2

Virtual Dictionary Extension

Lossless data compression is an important topic from both data transmission and storage point of view. A well chosen data compression technique can largely reduce the required throughput or storage need. As a tradeoff, data compression always requires some computing resources which are in correlation with the achieved compression rate. For a particular use case the best suitable compression method depends on the statistical characteristics of the data, the applied computing paradigm and the data access pattern.

2.1 Background – LZW Compression

2.1.1 LZW Encoding

During encoding LZW maintains a dictionary in which the entries are divided into two parts. The size and content of the first part, which is mostly called initial part, is immutable and contains all the individual symbols from a pre-defined alphabet with a sequence number associated with the position. The second part is a dynamic one and contains at least two symbols long words over the alphabet. The numbering of the dynamic part begins from the end of the initial part without overlapping. Supposing that our alphabet is the set of ASCII characters and we have an input text to be compressed, the dynamic part of the dictionary is built up according to the following rule, [48], [44]:

- The encoder builds words (W_b) from the input text character by character and looks up W_b in the dictionary.
- The encoder builds W_b until it is not available in the dictionary, or when the encoder reaches the end of the input text. When the W_b is not in the dictionary this means that W_b is one symbol longer than the previous longest character sequence with the same prefix W_{cm}. W_{cm} is also called *current match*.
- W_b will be written into the first empty position of the dynamic part of the dictionary. Alongside the encoder issues the sequence number of W_{cm}.
- Then the encoder forms a new W_{nb} from the last character of W_b.
- Then swaps W_b with W_{nb}, drops W_{nb} and starts a new cycle.

When the dictionary gets full the one of the most widely accepted strategy is that the dynamic part flushed out and rebuilt periodically to stay adaptive.

2.1.2 LZW Decoding

In case of decoding the decoder has to have the same initial dictionary. The decoder reads the issued sequence numbers. Based on the numbers and the static part of the dictionary the decoder is able to rebuild the dynamic entries. This information is enough to reconstruct the input text, [48], [44].

2.1.3 LZW, LZMW and LZAP Problems

As it is visible from section 2.1.1 the dictionary is built quite slowly. This means that the encoder can increase the stored entries by one character compared to the previously longest prefixes. In case when a relatively long substring occurs quite frequently, due to the dictionary construction strategy, the full coverage of that particular substring may require at least as many entries as long the substring itself is(Problem_1/P1).
The situation is even worse if two frequently occurring sub-strings(W_1, W_2) differ from each other only in the first character. In this case, due to the dictionary construction, full coverage of W_1 and W_2 may require twice as much entries in the dictionary as if W_1 and W_2 were identical (Problem_2/P2).
Besides the above two scenarios supposing that the encoder is in the middle of the encoding of an input text and there is a recurring substring W_1, the encoder will find that particular substring in its dictionary (and therefore compress the input text efficiently) only if it can start the word parsing from exactly the same character as did it in previous case. It means that an offset between the previous and actual substring parsing may significantly decrease the quality of the compression (Problem_3/P3).

Let us define *previous match* as the preceeding entry in the dictionary relative to current match.
LZMW(MW:Miller, Wegman) [44] tries to increase the hit ratio by inserting into the dictionary the concatenation of the previous match and current match. The main problem with this method is that it consumes the entries faster than LZW. Other problem is that encoding side time complexity is high compared to LZW.
LZAP(AP:All Prefixes)[44] is a derivative of LZMW and tries to resolve P1, P2 and P3 according to the following: during dictionary building besides the full concatenation of previous match and current match the extended previous matches are also stored. Extensions here mean all prefixes of the current match. That is why one match will occupy as many entries in the dictionary as many symbols reside in the current match. This approach can significantly increase the hit ratio, however it is too greedy from memory consumption point of view.

2.2 Virtual Dictionary Extension(VDE)

The goal is to eliminate somehow the memory consumption problem of LZMW or LZAP. To solve this problem a new approach will be introduced which I will call Virtual Dictionary Extension(VDE). VDE from processing point of view resides between LZMW

and LZAP. With Virtual Dictionary Extension we will be able to increase the hit ratio compared to LZW, but this method will require only as many entries as LZW.

To make it possible in the dictionary we have to distinguish the positions of the entries from their indexes/sequence numbers. In case of LZW, LZMW or LZAP the position of an entry is identical with its index. In those cases the distance between two adjacent entries is one. In the followings dictionary entries will be called *primary entries* and will be denoted by p. The idea is that in case of VDE the distance between two adjacent primary entries is one in terms of position but can be greater in terms of indexes. The position associated indexes will be denoted by i_p. The indexes which fall between two i_p will be denoted by i_v(virtual index). Virtual indexes, without position in the dictionary, refer to composite or virtual entries. That is why dictionary extension is called virtual. During encoding the indexes will be emitted instead of positions(as happened in case of LZW, LZMW or LZAP). The applied composition rule must consider that at decoding side we have to be able to reproduce the original input from the mixture of position associated and virtual indexes. Apart from this boundary condition we can choose any composition rule which fits to our problem domain. In the followings I will show the Linear Growth Distance(LGD) composition rule.

2.2.1 Linear Growth Distance Composition Rule

As previously mentioned the dictionary has an initial part and a dynamic part. Supposing that we have an alphabet which resides in the initial part of the dictionary. The initial part is immutable therefore in the followings we can consider it as a constant offset from both position and index point of view. To make the introduction of VDE-LGD encoding easier we ignore the initial part caused offset and focus only on the usage of dynamic part. In case of LGD we can count the position associated indexes according to the following formula:

$$i_p = \frac{p(p+1)}{2}, \tag{2.1}$$

which is nothing else but the triangular number [45]. Considering the linearly growing number of i_v between i_p, which is always equal with the number of preceding primary entries, with i_v we can refer to concatenations which are generated from words of previous primary positions. With this technique we can increase the hit ratio with identical number of entries.

Let's see an example: the text to be compressed is let's say: "*asdfasdr*". Based on the composition rule the following words will be available:

0	-	**as**,	**a**
1	-	**sd**,	**s**
2	-	asd	
3	-	**df**,	**d**
4	-	sdf	
5	-	asdf	
6	-	**fa**,	**f**

7	-	dfa	
8	-	sdfa	
9	-	asdfa	
10	-	**asdr**,	**asd**
11	-	fasdr	
12	-	dfasdr	
13	-	sdfasdr	
14	-	asdfasdr	

The primary entries are marked with bold. The emitted symbol itself is displayed after the comma instead of the index of the emitted symbol.

In case of any constraint regarding the maximum number of virtual words between two subsequent primary words is denoted by VDE-LGD(*max_constraint*). VDE-LGD or VDE-LGD(∞) is applied otherwise.

2.2.2 Linear Growth Distance Encoding

To explain encoding let us first compare the content of LZW(left column) and VDE-LGD(right column) dictionaries and the emitted indexes based on the previous example:

0	-	as,	a	0	-	*as*,	a	$\rightarrow$	i_p
1	-	sd,	s	1	-	s*d*,	s	$\rightarrow$	i_p
2	-	df,	d	3	-	d*f*,	d	$\rightarrow$	i_p
3	-	fa,	f	6	-	fa,	f	$\rightarrow$	i_p
4	-	asd,	as	10	-	asdr,	asd	$\rightarrow$	$i_v(=2)$

To determine the indexes let's consider the bold "asdr" row. In the legacy case "as" would be the current match. I propose to start examine after the "as"(marked by italic) match the successive primary entry without the first character, which is in this case "sd" without "s", that is "d"(marked by italic). In case of matching one takes the next primary entry, "df", and performs the previously mentioned examination again, "f"(marked by underline) in this case. However the next symbol in the input text to be encoded is "r", so the extension process stops here. When the last match has been reached it counts the Number of Hops(NoH) and maintains the first match. The index to be sent out will be computed according to the following rule:

- if the first match is the last match, so there is no subsequent match, the index is an i_p type and counted based on the dictionary position,
- if the first match differs from the last match the index to be sent is computed according to this:

$$i_v = i_p + (p_l - p_f),$$

where

– p_l is the position of last match, and

– p_f is the position of first match.

The original LZW algorithm requires the following modifications:

– First I have to introduce a new, so called, buffer area to be able to simulate and handle the subsequent word comparison failures. This solution makes it possible to continue the process in the middle of the "next entry", in case of comparison failure, without information loss.

– The second difference is that I have to distinguish from searching point of view the first match from subsequent matching(s).

– The third difference is that it has to differentiate the initial part of the dictionary from the dynamic part. In case of LGD virtual extension will be applied exclusively to the dynamic part of the dictionary.

2.2.3 Linear Growth Distance Decoding

At decoding side the reconstruction of the input works like the following: when an index arrives - denoted by i_a - the algorithm examines if it is a primary entry or not. To perform this the following formula is used:

$$p_c = \frac{-1 + \sqrt{1 + 8i_a}}{2}. \tag{2.2}$$

From here there are two main scenarios possible:

– In case when the p_c is an integer without remaining value this means that the dictionary entry searched for is a primary entry. It is possible to look up the entry from the dictionary directly.

– Otherwise take the floor function of the computed position, signed p_f. This will provide last primary entry of match. Then compute the base index from the position, signed with i_b, with the following formula:

$$i_b = \frac{p_f(p_f + 1)}{2}. \tag{2.3}$$

Then with a simple subtraction it is easy to define the NoH $= i_a - i_b$. With this information step back NoH and start to generate the derivative entry. From here, if the word is computed, the process continues as in case of the original LZW algorithm.

There is only a small difference compared to the original decoder method when the referenced primary entry still not present: it only can takes place when it depends on the previous primary entry. To compute the missing reference entry simply step back with NoH, which is practically 1 in this case. Then take the first character of that primary entry as an addition to the previously used entry, no matter if it is a derivative or primary one. Then this combined entry will be the missing referenced entry that have to be written into the dictionary. From here every step takes place according to has been written before.

The section relates to *Thesis 4.1.1*.

2.3 Complexities

Since the main goal of a compression method is to reduce the size of the input data, the theoretically available compression ratio is the most important factor. However, based on the usage pattern and constraints like computing or memory resource limitations, other factors also have to be considered. These factors are:

- encoding time complexity(encoding speed),
- encoding space complexity(encoding memory need),
- decoding time complexity(decoding speed),
- decoding space complexity(decoding memory need) and
- life-cycle of the compressed data(part of statistical characteristic).

These factors mostly depending on each other. In Fig. 2.1 the high level dependency is visualized.

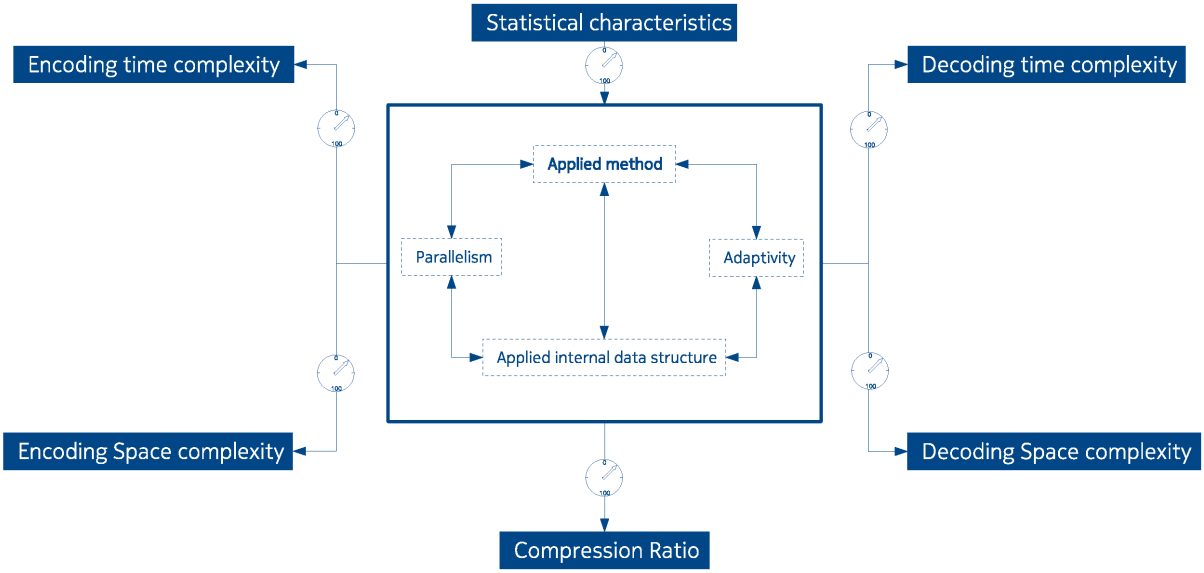

Figure 2.1: Dependencies and relations between the statistical feature of the data to be encoded, the achievable compression ratio and the space and time complexities.

There are existing analysis regarding LZ family like [46], [50] and [49], which are handling the question in general manner. However in the following the focus is exclusively on the boundary values comparison from both compression ratio, processing speed and memory need point of view.

2.3.1 Encoding Space Complexity

In this section the memory need of VDE-LGD will be analysed. To be able to perform this the capacity of the dictionary has to be distinguished from the actual size. The

relation between capacity and size is: size $\leq$ capacity. In this terminology size always refers to the actually occupied bytes in the buffer during encoding or decoding, so it is a dynamic descriptor. While capacity refers to the theoretically needed/achievable length if the input pattern is the most memory demanding from dictionary composition point of view, therefore it is a static descriptor. So the actual size always depends on the actually processed input pattern, while capacity is that size which is required if the most memory demanding input pattern would be processed.

LZW Encoding Capacity

In Fig.2.2 the encoding of the previous text is visible with LZW algorithm. Supposing that the initial alphabet is the ASCII table. Therefore the dynamic part of the dictionary starts from 256. Since in case of LZW the position is equal with the index the header sequence is continuous. Based on the dictionary composition rule the actually occupied space(the size) is marked by continuous line.

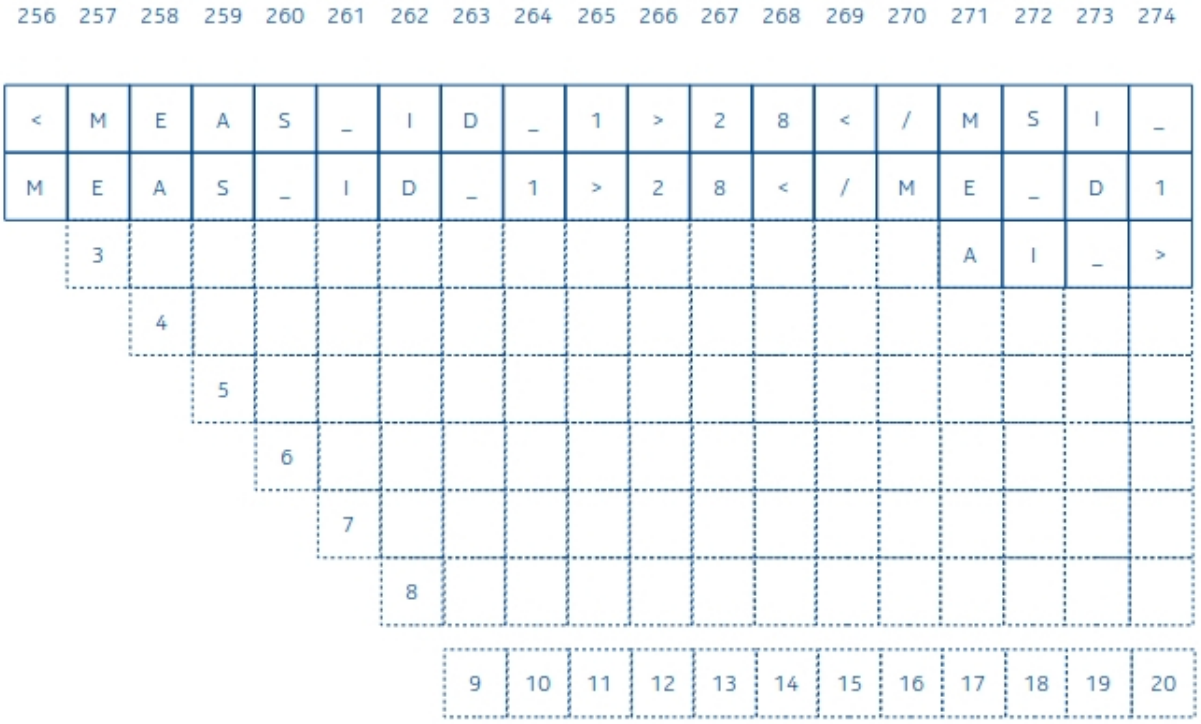

Figure 2.2: capacity vs size in case of LZW

As it is visible from the figure the occupied size is strongly pattern dependent. The dotted line marks the capacity of the dictionary. The worst case scenario from memory consumption point of view takes place when the input pattern, due to the construction of the dictionary, always makes possible to reuse the longest previously stored entry during encoding of the next portion from the input data. The pattern which meets these requirements is eg. the: "*aaaaaaa*...". Obviously this is a very rare pattern in case of compression, but gives us a baseline (this is the reference of a so called distortion factor). Naturally this pattern could be compressed by another representation like $n \times c$, where c is the repeating character and n is the repetition factor.

Now let's examine the capacity need of LZW. To be able to compare the growth rate of LZW to VDE a notation will be introduced below which is trivial in case of LZW, however will not in case of VDE. If the data to be compressed is a string of same characters and the initial alphabet resides from entry 0 to 255, then the newly attached characters (ac) can be expressed with the following formula:

$$ac_p = ac_{p-1} + 1 \mid p > 255, \tag{2.4}$$

$ac_{255} = 0$.
Entry (or word) level memory need is:

$$em_p = 1 + ac_p \mid p > 255. \tag{2.5}$$

Finally the position dependent aggregated memory need can be expressed by the following formula:

$$C_{LZW}(p) = 255 + \frac{(p-254)(p-253)}{2} \mid p > 255. \tag{2.6}$$

As it is visible from the formula the capacity need is depending on the number of allowed dynamic entries. The growth rate of the entries and the aggregated memory need is $O(p), O(p^2)$ respectively.
Let the length of the initial alphabet is S_{in}. Then the previous formula turns into the following one:

$$C_{LZW}(p) = (S_{in} - 1) + \frac{(p-(S_{in}-2))(p-(S_{in}-3))}{2} \tag{2.7}$$

VDE Encoding Capacity

Now the capacity need of VDE-LGD(∞) will be determined. In Fig. 2.3 the previous example is visible as it is processed by VDE-LGD(∞). In the figure the upper number sequence represents the positions, while secondary number sequence represents the position associated primary indexes. Just like before the actually occupied size is marked by continuous line. The way as the dictionary construction allows that the longest entry is more than one character longer than the second longest entry is clearly visible at position 271: The length of the longest stored entry is ten characters, while the second longest entries are two characters long.
The worst case scenario from memory consumption point of view takes place when the input pattern, due to the construction of the dictionary, makes possible the reuse of all the previously stored entries to encode the next portion from the input data. The prerequisite of this behaviour is that the first primary entry is not prefix of the second primary entry parallel the second primary entry is not prefix of the third primary entry. However both the lower positioned odd and even entries are prefixes of the higher positioned odd and even entries respectively. The pattern which meets these requirements is the "ababab..." alternating character sequence.

Let's see the entries if the algorithm is fed with the "ababab..." input. To make calculations easier the numbering of the dynamic part as will be shifted as the first position will be the zero.

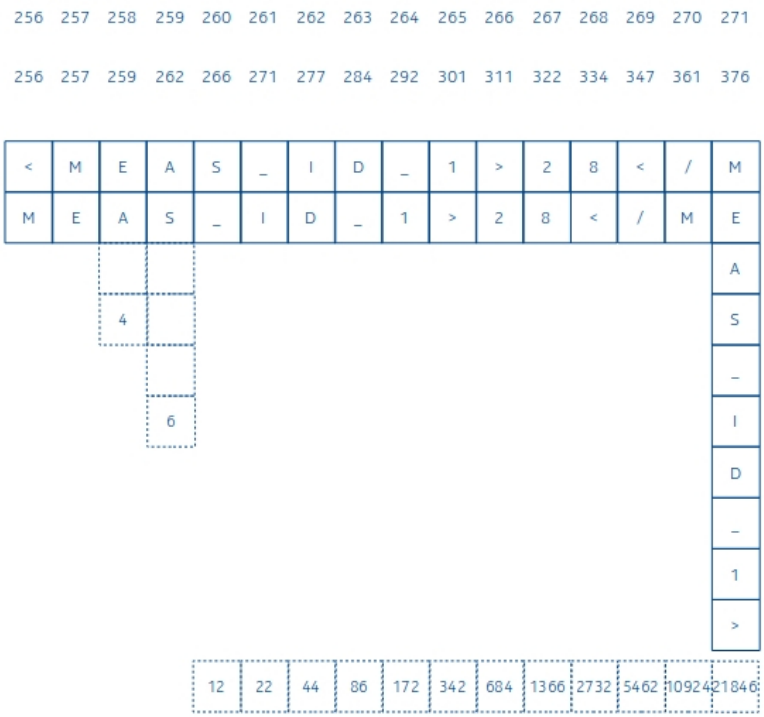

Figure 2.3: capacity vs size in case of VDE-LGD

0	0	1	a	b
1	1	1	b	a
2	3	3	a	bab
3	6	5	b	ababa
4	10	11	a	bababababab
5	15	21	b	abababababababababababa
6	21	43	a	bababababababababababababababababababababababab

The first column is the position, second is the associated primary index and the third column is the number of newly attached characters from the input. The recursive formula to express the position dependent newly attached (ac) characters is:

$$ac_p = (2 \times ac_{p-1}) + (-1)^p, \tag{2.8}$$

if $ac_0 = 1$ and $p = 1, 2, 3, \ldots$. In Fig.2.3 the dotted line rectangles contain numbers which greater by one than the numbers from the previous recursive formula. The results of the recursive formula are equal with the increments, while the numbers from the figure are equal by the occupied characters and due to the contribution of the dictionary the ending characters are stored twice since those are the starting ones of the succeeding entries. To be able to make calculations let's explicate the recursive formula:

$$ac_p = 2 \times ((2 \times ac_{p-2}) + (-1)^{p-1}) + (-1)^p, \tag{2.9}$$

$$ac_p = 2 \times ((2 \times ((2 \times ac_{p-3}) + (-1)^{p-2}) + (-1)^{p-1}) + (-1)^p, \tag{2.10}$$

which leads to the following series:

$$2^p - 2^{p-1} + 2^{p-2} - 2^{p-3} + 2^{p-4} - 2^{p-5} + ... \tag{2.11}$$

This series can be expressed with formula:

$$\sum_{i=0}^{p} 2^{p-i}(-1)^i \tag{2.12}$$

From the formulas above we can express the capacity need of primary entries and the aggregated dictionary as well:

$$em_p = 1 + ac_p = 1 + \sum_{i=0}^{p} 2^{p-i}(-1)^i \tag{2.13}$$

$$C_{lgd}(p) = S_{in} + p + \sum_{i=0}^{p+1} 2^{p-i}(-1)^i - (0^{1+(-1)^{p+1}}) \tag{2.14}$$

From the formulas it is visible that both the entry and the aggregated level growth rate is $C_{lgd}(p) = O(2^p)$ in contrast of $C_{LZW}(p) = O(p^2)$.

Least Memory Demanding Input Pattern

The least memory demanding input pattern is if the maximum length of the dynamic entries can grow by one character if and only if previously all the variations are stored in the dictionary from the preceding maximum length. The following example will expose what does it mean in practice.
Let the initial part of the dictionary is the first four letters from the English alphabet a,b,c and d. Then the following sequence of the letters will led to the least memory demanding entries: "*aabacadbbcbdccdda*". This sequence will led to the following structure in the dictionary (relative numbering):

01 - aa,	08-bb,	13-cc,	15-dd
02 - ab,	09-bc,	14-cd,	16-da
03 - ba,	10-cb,		
04 - ac,	11-bd,		
05 - ca,	12-dc,		
06 - ad,			
07 - db,			

Actually this is nothing else than all the pairs from the initial alphabet, which is the $V_n^{r,2} = n^2 = 4^2 = 16$. Here V refers to variation, $n = S_{in}$ is the cardinality of the alphabet, r in the upper index means that repetition is allowed and the number in the upper index refer to the length of the word over the alphabet. Of course the sequence can be continued

with all triplets $V_{S_{in}}^{r,3}$ and so on. The following formula expresses the aggregated number of entries if all the words stored with maximum m length:

$$\sum_{i=1}^{m} V_{S_{in}}^{r,i+1}. \tag{2.15}$$

To construct the least memory demanding sequence it is not enough to generate the increasing length variations: prefixes also must be avoided. In the following a construction method will be introduced over the previously shown four letters long alphabet. To generate the appropriate triplets the pairs will be systematically extended. During extension the first letter from the alphabet will be inserted in the middle of the existing pairs. Then the second letter from the alphabet and so on. The newly generated entries will look like this:

17-a(a)a,	24-b(a)b,	29-c(a)c,	31-d(a)d
18-a(a)b,	25-b(a)c,	30-c(a)d,	32-d(a)a
19-b(a)a,	26-c(a)b,		
20-a(a)c,	27-b(a)d,		
21-c(a)a,	28-d(a)c,		
22-a(a)d,			
23-d(a)b,			
33-a(b)a,	40-b(b)b,	45-c(b)c,	47-d(b)d
34-a(b)b,	41-b(b)c,	46-c(b)d,	48-d(b)a
35-b(b)a,	42-c(b)b,		
...			
65-a(d)a,	72-b(d)b,	77-c(d)c,	79-d(d)d
66-a(d)b,	73-b(d)c,	78-c(d)d,	80-d(d)a
67-b(d)a,	74-c(d)b,		
68-a(d)c,	75-b(d)d,		
69-c(d)a,	76-d(d)c,		
70-a(d)d,			
71-d(d)b,			

In the entries above the newly inserted characters are marked with parenthesis. From the entries the "original" input can be generated by concatenating the entries one after another with that constraint that during concatenation the first character of each entry has to dropped except the first entry:

"*aabacadbbcbdccdda*||*aaabaaacaaadababacabadacacadadaa*|*babbbabbbcbbbcbdbdba...*
...|*dadbdadcdadddbdbdcdbdddcdcdddddda*"

The double vertical line splits the different degree of variants. The single vertical lines split the string according the extension letter within the same degree of variants.
The generation can be continued in the following way: Take all the triplets and insert the first letter from the alphabet between the last and the last but one characters. Then the same insertion should be performed with all the remained letter from the alphabet.

The method will produce the least memory demanding input for LZW in case of unlimited number of entries and with some constraint can be applied to VDE as well.
Supposing that the initial alphabet is the extended ASCII table(256 characters), additionally the data to be encoded is the above created input string. Then the number of possible pairs in the dictionary is: $V^{r,2}_{S_{in}=256} = 65536$.
Regarding LZW this means that during encoding the first 65536 characters all the reference will refer to the initial/static part of the dictionary with uniform distribution in terms of frequency. In real implementations the number of entries are limited around 16000 entries due to space and time complexity constraint, so with the pairs only the dictionary can be filled with pairs only.
It was mentioned earlier that with some constraint the generated input can be applied to VDE as well. This constraint is the previously mentioned limited number of entries. Since during pairs the prefix free words can be ensured, however when longer words are generated the above method will result overflowing, which behaviour overrules the initial condition that the dynamic entries can grow by one character if and only if previously all the variations are stored in the dictionary. But this would require more than 64K primary entries, which is not valid in actual implementations.

2.3.2 Compression Ratio

In this section the relation between the length of dictionary entries and the theoretically achievable compression ratio will be exposed. First LZW will be examined then it will be compared with VDE-LGD(∞).
These additional notations also will be used during the examinations: let S_t is the total number of entries and $S_d = S_t - S_{in}$ is the maximum number of dynamic entries. Denote $R_b = \lceil log_2(S_t) \rceil$ the required number of bits to represent the full dictionary.

LZW Compression Ratio

Supposing that the initial dictionary of LZW is the eight bits ASCII table. Let an additional constraint that maximum 256 dynamic entries are allowed. This means that the range from 0 to 511 has to be covered unique entry positions $S_t = 512$. The required number of bits to be able to represent a particular entry is $\lceil log_2(n) \rceil$, where n is the position of the entry. It is visible in Fig.2.4. However in practice the above number

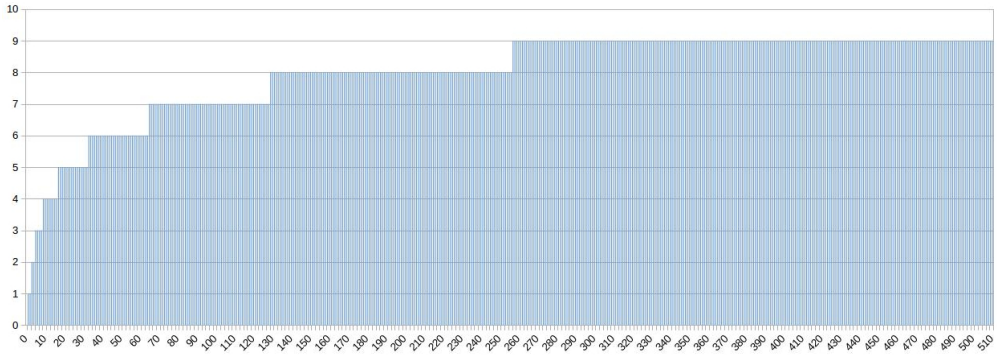

Figure 2.4: LZW non uniquely decodable representation need [bit]

of bits do not ensure the unique decodability need. In this particular case $R_b = 9$. Supposing that the input pattern is most memory demanding $n \times c$ type. Then the entry level compression ratio can be determined. This is the division of the representation need of the entry by the length of that particular entry as it is visible in Fig.2.5.

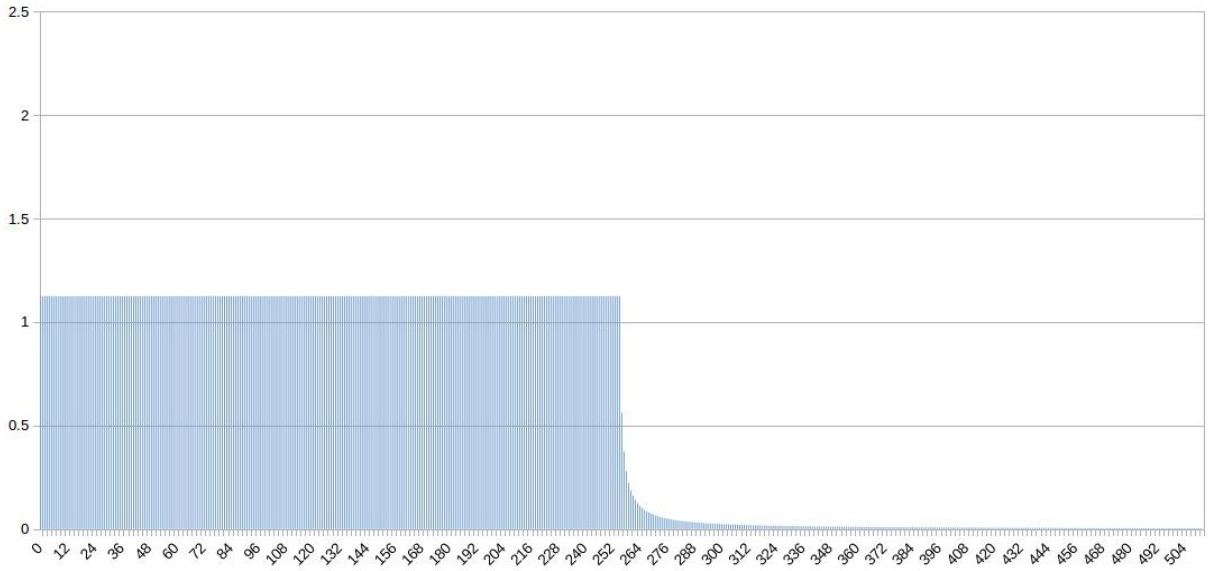

Figure 2.5: LZW entry level compression ratio

The importance of this figure is that it point out to the theoretically achievable lowest compression ratio and the dynamic behaviour of the algorithm. With the given conditions the lowest compression ratio is:

$$CR_{LZW} = \frac{R_b}{\lceil log_2(S_{in}) \rceil (em_{S_d} + 1)}, \tag{2.16}$$

where em_{S_d} refers to the ${S_d}^{th}$ dynamic entry, which is 256 in this case.
With term *"final match"* that operation will be denoted when the algorithm find the longest fitting entry from the dictionary to the next portion of the input data. During encoding the first character the first match is the final match since only single characters reside in the dictionary. As the dictionary growing probably several additional matches will follow the first matches before the final matches (,otherwise the input is the sequence of the letter from the alphabet with that length which is the size of the alphabet, or the dictionary dynamic part is too small). Every final match is followed by a dictionary identifier printout. Let the number of final matches is f_m. With this special input pattern during dictionary construction the compression ratio continuously get better. When the dictionary full the final matches always refer to the last entry therefore the compression ratio tends to:

$$\lim_{f_m \to \infty} CR_{LZW}(f_m) = \frac{R_b}{\lceil log_2(S_{in}) \rceil (em_{S_d} + 1)} \tag{2.17}$$

According to the formulas the highest memory usage can lead to the best compression ratio. Of course there are techniques which can reduce the actual memory need but on the other side those techniques increase the time complexity of the algorithm.

Now let the input data pattern is the least memory demanding one. In this case supposing that the number of dynamic entries are limited between 1 and 65536, that is $1 <= S_d <= 65536$. Due to the input during the construction of the dictionary always the simple letters from the static part will be referred. Therefore the compression ratio is the following during construction time:

$$CR_{LZW}(1 <= f_m <= S_d) = \frac{R_b}{\lceil log_2(S_{in}) \rceil} >= 1. \tag{2.18}$$

If the maximum number of dynamic entries $S_d = S_{in}^2$ then

$$\begin{aligned} CR_{LZW}(1 <= f_m <= S_d) = \frac{R_b}{\lceil log_2(S_{in}) \rceil} = \\ = \frac{\lceil log_2(S_{in} + S_{in}^2) \rceil}{\lceil log_2(S_{in}) \rceil} = \frac{\lceil log_2(S_{in}(1 + S_{in})) \rceil}{\lceil log_2(S_{in}) \rceil} = \\ = \frac{\lceil log_2(S_{in}) + log_2(1 + S_{in}) \rceil}{\lceil log_2(S_{in}) \rceil} \approx 2. \end{aligned} \tag{2.19}$$

Supposing that the number of dynamic entries tends to infinite. Due to the construction of the input the dictionary, if the length of the words incremented by one character during the processing of the portion of input the references always will refer to the words from the previous range, where the words are one character shorter. However the representation need R_b increased by one bit. Therefore the compression ratio is always greater than one.

VDE Compression Ratio

Now let's see the same analysis of VDE-LGD. Let the initial dictionary is the eight bits ASCII table again. The restriction for the number of primary entries is just like in the previous case: $S_d = 256$. In contrast to LZW this mean the range from $S_{in} : 0 - 255$ and $S_{de} : 256 - 33152$ has to be covered, where S_{de} refers to the extended dynamic entries. Therefore $S_t = S_{in} + S_{de}$. Choosing the previous representation type the required number of bits is $R_b = 16$.

The first ten primary and related virtual indexes will be visible in Fig.2.6. The dark columns sign the position associated primary entries. Vertical axis represents the length of the stored strings in bytes. Actually only the position associated strings will be stored, but it is possible to refer that strings which are start with a primary entry and fully cover one or more succeeding primary entries(these are the virtual indexes). In the figure the numbering starts from 256 as the starting point of the dynamic dictionary.
Based on the previously introduced storage and representation need it is possible to determine the theoretical entry level compression. In Fig. 2.7 both the primary and virtual index related compression ratio is visible. It means that only twenty three primary entries are needed to cover the first 256 dynamic entries due to the extension. From the figure it is visible that the compression ratio of VDE-LGD(∞) tends much more faster to zero than LZW. Let's compare the theoretical compression ratios at primary entry 23 from the dynamic dictionary, which would be printed out during the 24^{th} final match, $f_m = 24$. In case of LZW $R_b = 9, em_{23} = 24$ and $\lceil log_2(S_{in}) \rceil = 8$, that is:

$$CR_{LZW}(24) = \frac{9}{24 \times 8} = \frac{3}{64}, \tag{2.20}$$

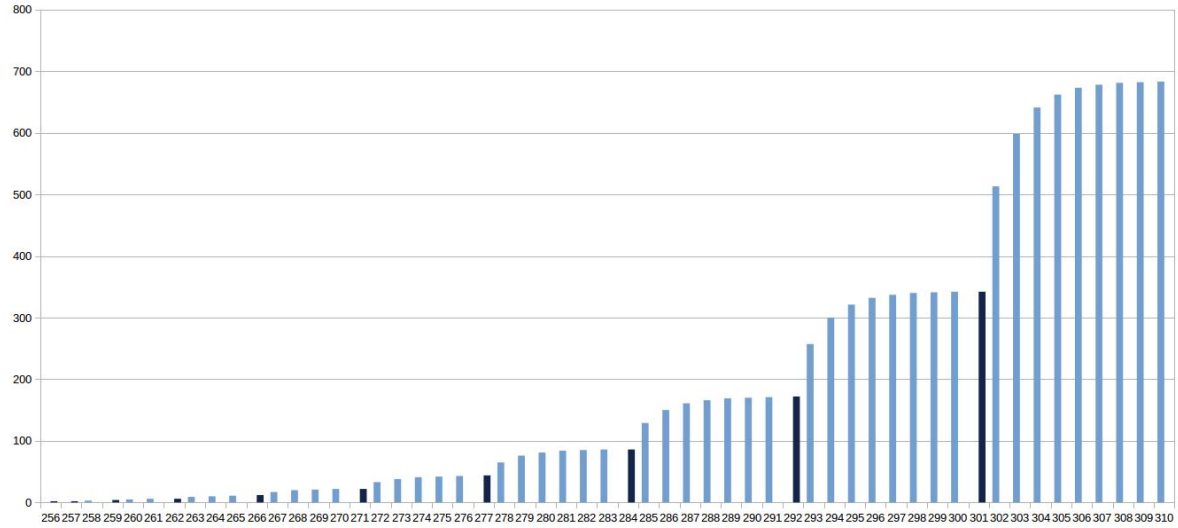

Figure 2.6: VDE theoretical expansion. Vertical axis represents the length of the stored strings in bytes. Horizontal axis represents the indices, the dark columns sign the position associated primary entries.

while in case of VDE-LGD $R_b = 16, em_{23} = 2796204$ and $\lceil log_2(S_{in}) \rceil = 8$, that is:

$$CR_{VDE}(24) = \frac{16}{2796204 \times 8} = \frac{1}{1398102} = \frac{3}{4194306}. \tag{2.21}$$

This comparison shows the fact that with VDE significantly better compression ratio can be achieved than LZW; of course this is very input pattern dependent.

From the other side if VDE would be fed with an $n \times (a)$ input pattern then it would behave like a traditional LZW from memory consumption point of view. However this would led to more poor compression ratio since growth rate of primary entries is $O(n^2)$, while $O(n)$ in case of LZW. Therefore, apart from the uniquely decodable representation, encoding of $n \times (a)$ input pattern with VDE the compression ratio would be twice as much as with LZW. Let n is the n^{th} primary index. Since the length of associated entries are equal the following equation is true:

$$CR_{VDE} = log_2(n^2) = 2log_2(n) = 2CR_{LZW}. \tag{2.22}$$

Considering the uniquely decodable representation the result will be very close to the theoretical value:

$$CR_{VDE} = \frac{16}{9} CR_{LZW} \approx 2CR_{LZW}. \tag{2.23}$$

This means that VDE fulfills that expectations if it should provide the asymptotically optimal feature regarding worst case scenario.

The worst case scenario will be given to the limited least memory demanding input pattern. Supposing that $S_{in} = 256$. Then VDE also could have $S_d = 65536$ primary entries. In this case $S_{de} = \frac{65536 \times 65537}{2} = 2147516416$. Based on this $R_b = \lceil log_2(S_{in} + S_{de}) \rceil = 32$.

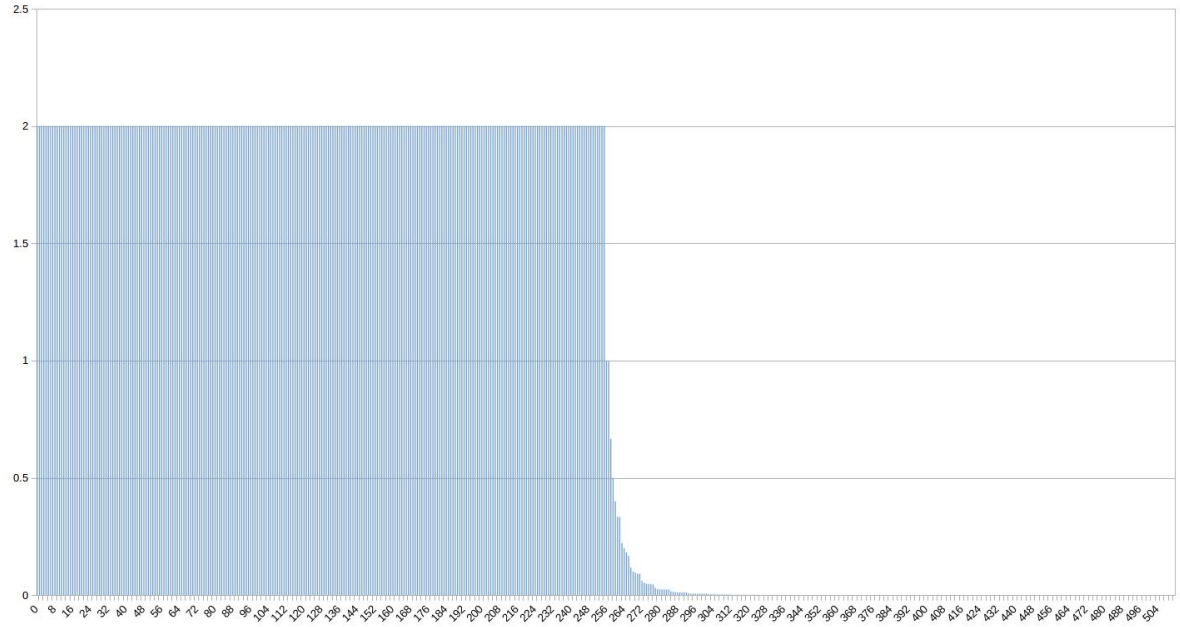

Figure 2.7: VDE theoretical entry level compression. Vertical axis represents the compression ratio. Horizontal axis represents the indices.

During dictionary construction every printout would lead to the following compression ratio:

$$CR_{VDE}(f_m) = \frac{32}{9} \approx 4 \approx 2CR_{LZW}, \tag{2.24}$$

as in previous case.

2.3.3 Encoding Time Complexities

In this section the LZW encoding speed will be compared to VDE-LGD(∞) encoding speed during dictionary building. The dependency from the length of the dictionary also will be examined. In the followings the time complexity will be determined as the total cost which is required to process the input characters. Considering encoding speed let's identify the types of the operations and the associated sub-costs:

- read a word from the input (c_{re}); where one character represents the shortest word,
- search for the longest matching word from the dictionary (c_{sea}),
- determine the output value (c_{de}),
- write out the value (c_{wr}) and
- insert the new word into the dictionary (c_{ins}).

The costs of these steps will be the basis of the analysis.
To make further calculations easier supposing that read a word from the input or write out the determined output value can be considered constant and equal, that is $c_{re} = c_{wr} =$

$const_1$.
The realization of the dictionary is usually a kind of associated array. However the cost of the search and the modification of the association array is the applied data structure dependent operation. Mostly hash table or a sort of binary search tree(red-black tree, b-tree, etc.) is applied as an associative array. In this case the application of a prefix-tree is also possible.
Supposing that the length of the dictionary is free to choose, however once it is determined will not change during the encoding process. In this case hash table could be a good choice, since if the length is preliminary known then the time demanding re-hashing is avoidable. Additionally it can provide for both search and insertion that the average cost of these operations is O(1). Therefore in the following examination a hash table will be the dictionary, additionally search and insertion related costs can be considered $c_{sea} = c_{ins} = const_2$.

LZW Encoding Time Complexity

According to the theory of LZW it can encode the input character by character. It means that every character read has a $const_1$ cost.
During search the encoder always goes to the first fail, which means that particular characters will be looked up twice. The relative frequency of duplicated comparisons is in relation with the average entry length in this case. The number of duplicated lookups is limited to the number of dynamic entries in the dictionary which is S_d.
The determination of the output value is simple in this case and does not require any complicated computation. Therefore this operation can be considered as a constant duration operation with cost $const_3$.
Let the number of input characters is n.
First the $n \times (a)$ input pattern and its processing time will be examined. The cost of the reads is equal with the number of the input characters, which is: $T_r = n \times const_1$.
During the encoding of this pattern the relative frequency of duplicated comparisons is a linearly decreasing function. Therefore theoretically the weight of this charge is tend to zero. In fact there is a practical limit, which is influenced by the number of entries. During dictionary construction n characters are divided into p linearly growing length entries, where p can be determined according to the following formula:

$$p = \lceil \frac{-1 + \sqrt{1 + 8n}}{2} \rceil. \tag{2.25}$$

Therefore the cost of comparisons:

$$T_{comp}(n) = (n + p)const_2. \tag{2.26}$$

The number of c_{de} and c_{wr} is equal with p, therefore:

$$T_{ins}(n) = p \times const_2,\ T_{de}(n) = p \times const_3,\ T_{wr}(n) = p \times const_1. \tag{2.27}$$

So total cost is:

$$T(n) = T_r(n) + T_{comp}(n) + T_{de}(n) + T_{ins} + T_{wr}(n). \tag{2.28}$$

The formula points out to the dependency from the input statistical characteristics: if the input can be compressed with the highest efficiency then $p/n \to min$ and thus

$T(n) \to min$.

Examining the worst case CR_{LZW} pattern it is clear that $p/n \to max$, that is $T(n) \to max$.

VDE Encoding Time Complexity

Considering the VDE encoding method two different phases can be differentiate. During first phase VDE works just like LZW: reads a character from the input, attaches it to the word to be searched for in the dictionary. When the algorithm cannot find more identical entry the second phase wil start. It examines if the succeeding entry of the last match is identical with the upcoming part of the input. To perform this examination two strategies can be applied. First strategy does not take into account the length of the succeeding entry and keeps on the reading and comparison character by character. Another strategy takes into account the length of the succeding entry and reads the required amount of characters from the input. During further examinations the second approach will be applied. Let the input size again n.

Supposing that $o = (n/2)$ and the input pattern is the $o \times ab$, which has the most favourable compression ratio. In this case the n number of characters are splitted into p number of positions:

$$p = [\log_2(3n)] + 1, \ if \ n \geqq 2. \tag{2.29}$$

In this case the second phase is the "normal" operation from the beginning. This means that during processing the input the number of read and comparison operations has a quadratic proportion with the number of used positions:

$$T_r(n) = p^2 \times const_1, \ T_{comp}(n) = p^2 \times const_2. \tag{2.30}$$

Determination of the output composed of the sub-cost a subtraction, a multiplication and an addition. These costs are depending on the number of digits of the operands. However, due to the special pattern, the output numbers are strictly monoton growing ones. These have logarithmic proportional connection with the number of digits. The most expensive operation is the multiplication since it has a quadratic dependency from the digits of the operands:

$$T_{de}(n) = O(\lceil log(S_{in} + p^2) \rceil^2), \tag{2.31}$$

and $T_{wr}(n)$ and $T_{ins}(n)$ just before:

$$T_{wr}(n) = p \times const_1, \ \ T_{ins}(n) = p \times const_2. \tag{2.32}$$

So,

$$T(n) = T_r + T_{comp} + T_{de} + T_{ins} + T_{wr}. \tag{2.33}$$

Examining the worst case CR_{VDE} pattern it is clear that $p_{VDE} = p_{LZW}$, and therefore $T_{VDE}(n) \to max$.
Examining the execution times it is easy to see that considering the most favorable patterns VDE can more efficiently encode a unit of input data than LZW.
However if the input pattern is the worst case from encoding point of view, then no significant difference between the two method: the lack of squaring does not increase the time complexity of VDE.

Decoding Time Complexity

During decoding the main operations are the read an identifier, decode, insert new entry into the dictionary and printout the word. The search operation is missing. It is easy to recognize that the difference between LZW and VDE is that in case of VDE square root computation is required to be able to lookup the portions of the decoded data. However as a result of this operation several read operations may be avoided. The question where is that limit when the computation of square root more efficient than the substituted read operations?
In case of most favorable pattern, due to its monotone characteristic, there will be a theoretical threshold when it is worth to substitute the reads with computation, since its normalized gain gets higher and higher.
In case of worst case pattern VDE will perform more poor than LZW since it can identify only one character, while must compute the square roots for each and every virtual index.

The section relates to *Thesis 4.1.2*.

2.4 Quantity Analysis on the Chaining of Repetition Free Words Considering the VDE Composition Rule

The motivation behind this section was raised during the examination of the boundary value patterns of the Virtual Dictionary Extension (VDE) compression method. During the VDE compression method *words* are stored in the so called compression dictionary, therefore the base unit of the boundary value pattern is the *word*.

2.4.1 Terms and the Formal Definition of the Goal

Terms

Let A an *alphabet*, which is the finite set of symbols. We can define *words* over A. Words defined over A are sequences of symbols from A. In the following the symbol, character or letter will be used as a synonym of each other. Let's define the following sets of words:

- A^* refers to such a set of words in which empty word or word with zero length also available,
- A^+ refers to such a set of words in which the length of the words is at least one character long,
- A^q set contains all the q length long words.

The q length of a w_i word is equal with the number of letters in w_i, eg.: $q = |w_i|$. $Factor$ of a w_i word is a block of consecutive letters of w_i. Under $prefix$ of a w_i word I mean such an u factor of w_i, where u consists of identical sequence of identical characters of w_i on the first $|u|$ letters. Therefore $|u| \leq |w_i|$.

Considering the $A = \{a, b\}$ alphabet. The number of different q characters long words which can be composed over A, is equal with the $b = 2$ based exponential function, that is b^q. According to the LZW compression method the so called *dictionary* is composed and maintained during the *encoding procedure*. The dictionary consists of w_j words from the set A^{2+} and the word associated identification numbers. These numbers are simple sequence numbers. During compression these sequence numbers are written out instead of the associated words. According to the composition rule of the dictionary there must exist a w_i word, with length $(q-1) = |w_i|$, in the dictionary before any w_j word, with length $q = |w_j|$, can be written into it, where w_i must be the prefix (or fraction) of w_j. Additionally the first character of w_j must be identical with the last character of $w_{(j-1)}$.

Definition 2.4.1. These identical characters of one after another words will be called **shared characters**.

The requirement of the existence of *shared characters* is called to **connectivity condition**. The set of words which satisfy the *connectivity condition* are the **connectible words**. The composition rule provides that the dictionary is duplication free: any word can be placed only once into it. We can easily determine the input character sequence to achieve the poorest compression ratio: the shortest words should be associated to the smallest sequence numbers.
Consider a dictionary where to each and every entry the number of associated identifications is equal with the position of that particular entry within the dictionary. So, one identifier/sequence number is associated to the first entry, two identifiers/sequence numbers are associated to the second entry, etc. With this association rule the concatenation of one after another words are also can be unequivocally identified over the individual entries[1]. During the concatenation based word identification the *shared characters* will be considered once per boundary.

Definition 2.4.2. Those words which are directly written into the dictionary will be called **primary words**, and denoted by w_p.

Definition 2.4.3. The composition based words which contain w_p words as $factors$, will be called **virtual words**, and denoted by w_v.

Let's denoting by D the set of those words, which are still identified in the dictionary, no matter if the identified word is primary or virtual.

Definition 2.4.4. For a given q length, those *primary words* for which $w_{p_i} \in A^q \setminus D^q$ is true are called **available words**.

Formal Goal

The goal is to determine the features of such a string where the shortest *primary words* occupy the positions with smaller sequence number considering that the presence of these *primary words* based *virtual words* may modify the available number of longer *primary words*.
Formally I would like to determine the features of a w_{v_n}, which is composed from *primary words* on the way

[1] This method in itself let the occurrence of duplicate entries.

– that $|w_{p_i}| <= |w_{p_j}|$ is true $\forall i, j \in \mathbb{N} \mid i < j$, and $w_{p_i} = w_{p_j}$ then, and only then if $i = j$,

– $w_{v_l} = w_{v_m}$, then and only then if $l = m$, and

– additionally the presence of $w_{p_x}^{q+1}$ implies that there are words $w_{p_i}^q, w_{p_{i+1}}^q, ..., w_{p_{i+m}}^q \in A_p^q$ and $w_{v_j}^q, w_{v_2}^k, ..., w_{v_l}^q \in A_v^q$ where $i <, .., < i + m < x,\ j < x,\ k < x,\ l < x$ so that $A_p^q \cap A_v^q = \varnothing$ and $|A^q| = |A_p^q \cup A_v^q| = b^q$.

2.4.2 Quantity Analysis of Primary Words Influenced by Virtual Words

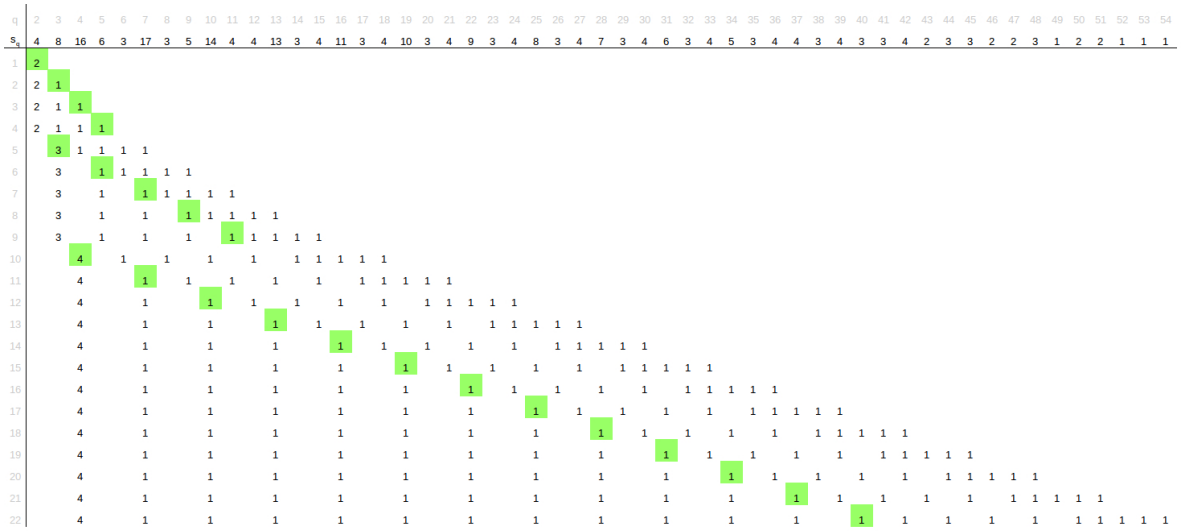

Figure 2.8: Implicit dependency up to four characters long *primary words*. Numbers greater than '1' represent the *primary words*. Numbers marked with '1' represent the presence of *virtual words*. Green squares are the envelopes of the direct effect of the homogeneous concatenations of q characters long words.

Supposing that we have an alphabet $A = \{a, b\}$. The cardinality of A, denoted by b, is two. Composing all the two letters long words will result $(b = 2)^{q=2} = 4$ words. Since there is no preliminary constraint, origins from existing *virtual words*, the words can be connected to each other by multiple order. As a result these two-letters-long words will compose three three-letters-long, two four-letters-long and one five-letters-long *virtual word* by all means. Supposing that there is at least one way as the five uncovered three-letters-long *primary words* are connectible. Then these connected *primary words* (that is written into the dictionary) will cover (that is, can be conformed to) four five-letters-, three seven-letters-, two nine-letters- and one eleven-letters-long *primary words*.

The Effect of the Prime Numbers

To determine whether a t characters long word can be a composition of two subsequent *primary words* with length q, where $q \leq t$, we can use the following formula, considering the rule of the dictionary construction:

$$q = \frac{t+1}{2}. \tag{2.34}$$

In case of the result is an integer then the concatenation of two subsequent q characters long *primary words* implicitly cover another $t = (2q - 1)$ characters long *primary word*. Those $t = (2q - 1)$ characters long words, which are implicitly generated through the concatenation of shorter (q characters long) *primary words* preceding the usage of the same t characters long *primary words*, are an examples of the previously introduced *virtual words*. Therefore the *virtual words* and the *implicitly covered entries/words* will be used as a synonym of each other.
In case of we are interested about if is there a t characters long *virtual word* which is the concatenation of three, q characters long, subsequent *primary words* we have to examine if the $q = \frac{t+2}{3}$ value is an integer or not? Generally speaking, by denoting the number of concatenated, subsequent, same length entries with x, the formula turns into the following one:

$$q(x) = \frac{t + (x - 1)}{x}. \tag{2.35}$$

With a small transformation we will get the following formula:

$$q(x) = 1 + \frac{t - 1}{x}. \tag{2.36}$$

The formula will lead to integer results, that is word with valid length, in each and every case when the counter is divisible by the denominator. Since the negative length is not an option, the value of x can be greater or equal than 1 and less or equal than $t - 1$:

$$1 \leq x \leq (t - 1). \tag{2.37}$$

Substituting the boundary values into the formula:

$$q(1) = 1 + \frac{t - 1}{1} = t, \tag{2.38}$$

$$q(t - 1) = 1 + \frac{t - 1}{t - 1} = 2. \tag{2.39}$$

This means that a t characters long entry without concatenation can cover a t characters long entry, which is trivial. On the other hand $(t - 1)$ two characters long strings has to be concatenated to cover a t characters long string. The point is if the counter is a *prime number*, that is, $t = (prime\ number) + 1$ then the denominators can only be one of the above mentioned boundary values. This means that if the number of two characters long entries is less than $t - 1$, then there is no theoretical way to cover the t characters long *primary words* as the concatenation of $q < t$ long *primary words*. Of course, the concatenation of two *primary words* with different length can cover a t characters long *primary word* even though $(t - 1) = prime\ number$. However, we expect negligibly small amount from these type of constructions, otherwise the third condition from *subsection 2.4.1* would be significantly hurt. Additionally, the mass appearance of such type of *virtual words*, which are the outcome of the connection of words, with different lengths would result even more negligible effect on the available amount of longer words, due to the exponential growth rate, as we shall see.

Our task is to determine whether how many n characters long *primary words* remain from the 2^n ones, if we consider the concatenation of homogeneous l, m, etc. characters long words, as *virtual words*, according to the formula (2.36).

Theorem 2.4.1. From the b^q pieces of q characters long *primary words* there are no more than $c(b)\sqrt{b^q}$ ones can be covered by the previously generated *virtual words*.

Here c is a constant next to a given $b = |A|$.
From the field of number theory it is known that for every natural number n,

$$d(n) \leq 2\sqrt{n}, \tag{2.40}$$

where $d(n)$ is the number of divisors of n. Other notation of $d(n)$ is $\sigma_0(n)$.

By now the connection between the *Theorem-2.4.1* and the theory of prime numbers is clear, and the expectation become obvious regarding the asymptotically vanishing ratio of influencing *virtual words* on a given length of *primary words*.

Proof of the Square Root Proportional Influence of Virtual Words

By focusing exclusively to the quantity based coverings I can determine the exact number of *virtual words* according to the algorithm described below. But it is important to mention that this algorithm does not take into account the number of *shared characters* and some other rules, which are influencing the possible number of connectible words. The outcome of this algorithm is an array. In this array the columns are associated to the different length of words irrespectively if the words are primary or virtual ones. The first column represents the two, the second column represents the three and so on characters long words. Therefore in a particular column the maximum number of the filled cells is equal with b^q where q means the $n^{th} + 1$ column. The header of this array contains the number of the already filled cells in the given column (not the sum of their values), denoted by s_q. Initially the header contains infinite number of zeros, and according to our notation $s_q \leq b^q$. A particular cell in the array is identified by it's row and column numbers respectively, and denoted by $a_{i,j}$. Initially the cells are just left blank. The array filling always starts from the $(1, 1)$ element. To be able to localize the position of the so called base element during the filling I will introduce two pointers:

- let r the position of the actual row (row under write), and
- let the previously mentioned q the position of the $actual\_column + 1$.

According to the composition rule of *virtual word* a stand alone *primary word* is not able to cause any covering. Therefore we simply write the column associated length of the word into $a_{r,(q-1)} = a_{1,1}$, which is two according to $q = 2$. Parallel we increase the value of $s_q (= s_2$ in this case) by one. Then, as just in case of real dictionary composition, we have to increase the row pointer by one. Since the $s_q \leq b^q$ relation is still valid we can write number q into the cell $a_{i=r=2,j=q-1=1}$. However, two *primary words* will compose a *virtual word*. It is taken into account according to the following manner: due to the composition rule the number of *virtual words* should be exactly $r - 1$. Since $r = 2$ the number of virtual entries is one in this case. Now we have to find the position of the *virtual word* in the array. We will administer the effect of a primary entry in the row of that entry. Two two characters long entry can expose its effect in the three characters long entries. Generally speaking we will add the values of the cells in the $(q - 1)^{th}$ column and deduct the result with the number of $factors$ / *primary words* minus one. In this particular

case $a_{1,1} + a_{2,1} - 1 = 3$. Let 1 is notation of the *virtual words* in the array. Then we will write $a_{2,2} = 1$ and parallel increase the $s_{q=3}$ counter. In case of $s_q > b^q$ we will increase the value of q by one. However, from here during the summation we have to take into account that there are shifts in the columns of *primary words*. Due to the composition it is not possible that any indication of a virtual entry appear prior to a primary entry. Therefore during summation we start from the $j = q - 1$ column and check in each and every step if is there any value, which is greater than one at cell $a_{i,(j-1)}$. If so then we have to decrease the value of j by one. To be able to distinguish the pointers belong to the base position and therefore to primary entries from pointers which are required for computing the positions of virtual entries we will introduce the following utility and virtual pointers:

- r_u is the row utility pointer,
- q_u is the column utility pointer + 1,
- q_v is the column position + 1 of a virtual entry.

Additionally n is used to store sub-totals during the process.
Part of the above algorithm generated semi-infinite array is visible in Fig.2.8. In this example $b = 2$ has been selected, but of course this value can be an arbitrary one. In the figure the words up to four characters are represented. It is visible from the figure that, as it is expected, all the two-characters- long words appear as primary entries. As a consequence three three-characters-long word appear as *virtual words*. Therefore only five of the eight three-characters-long word can appear as *primary word*. As just in case of four-characters-long words: three of them is covered by the *virtual words* and only thirteen can appear as primary entry.
The number of *primary words* with a given q length will be denoted by $s_p(q)$. In this case the the following inequality is true: $s_p(q) <= b^q$. By introducing the $s_v(q)$, which is the number of *virtual words* belong to a given q, we can write that:

$$s_p(q) + s_v(q) = b^q. \tag{2.41}$$

By examining if which $q' \leq q$ can influence, that is decrease the number of q characters long *primary words* consider the formula (2.36). By performing the $t \to q$ substitution we can say that the length of longest influencing q' *primary word* is:

$$q' = \frac{q+1}{2} \tag{2.42}$$

if q is odd.
As long as q is even the $q + 1$ is odd, therefore in this case the longest possible *virtual word* must consists of at least three concatenated q'' *primary words* with the length:

$$q'' \leq \frac{q+2}{3}. \tag{2.43}$$

Supposing that $s_p(q') = b^{q'}$, than $s_v(q) \geq b^{q'} - 1$. In case of q is even then $s_v(q) \geq b^{q''} - 2$. Supposing that all the q' is available, that is $s_v(q') = 0$ (uncovered by such *virtual words*

which are the result of the concatenation of shorter *primary words*). Then we generally can say that

$$\begin{aligned} s_v(q,x) &= b^{\frac{q+x-1}{x}} - (x-1) \\ &= \sqrt[x]{b^{q+(x-1)}} - (x-1) \\ &= b^{q'} - (x-1) \end{aligned} \tag{2.44}$$

Here x indicates the number of concatenated *primary words*, where the length of these *primary words* is obviously shorter than q. Based on (2.44):

$$s_v(q) \leq \sum_{i=2}^{q-1} b^{\frac{q+i-1}{i}} - (i-1). \tag{2.45}$$

Applied by the geometric progression sum formula,

$$(b-1)\sum_{i=1}^{k} b^i = b^{k+1} - b, \tag{2.46a}$$

$$\sum_{i=1}^{k} b^i = \frac{b^{k+1}-b}{b-1} \tag{2.46b}$$

we will get, depending on if q is odd or even:

$$\sum_{i=2}^{q'} b^i = \frac{b^{q'+1}-b}{b-1} - b = \frac{b^{\frac{q+1}{2}+1}-b}{b-1} - b = \frac{b^{\frac{q+3}{2}}-b}{b-1} - b \tag{2.47a}$$

$$\sum_{i=2}^{q''} b^i = \frac{b^{q'+1}-b}{b-1} - b = \frac{b^{\frac{q+2}{3}+1}-b}{b-1} - b = \frac{b^{\frac{q+5}{3}}-b}{b-1} - b \tag{2.47b}$$

respectively. Since the formulas above contain all the words with length $2 \leq i \leq q'$, these values must be decreased by the number of words with given length minus one, as I noted it in (2.44). Depending on if q is odd or even the formulas are the followings respectively:

$$s_v(q) \leq \sum_{i=2}^{q'} b^i - \sum_{i=1}^{q'-1} i, \tag{2.48a}$$

$$s_v(q) \leq \sum_{i=2}^{q''} b^i - \sum_{i=2}^{q''-1} i. \tag{2.48b}$$

Based on (2.41):

$$s_p(q) = b^q - s_v(q). \tag{2.49}$$

In case of b is fixed and q is odd:

$$\begin{aligned} s'_p(q) \geq b^q + b - \frac{b^{\frac{q+3}{2}}-b}{b-1} + \frac{q'(q'-1)}{2} - 1 = \\ b^q + b + \frac{b}{b-1} + \frac{1}{2}((q')^2 - q') - 1 - \frac{\sqrt{b^3}\sqrt{b^q}}{b-1}, \\ s'_p(q) \geq b^q + c_1 + c_2((q')^2 - q') - c_3\sqrt[2]{b^q}. \end{aligned} \tag{2.50}$$

In case of b is fixed and q is even:

$$\begin{gathered} s''_p(q) \geq b^q + b - \frac{b^{\frac{q+5}{3}} - b}{b-1} + \frac{q''(q''-1)}{2} = \\ b^q + b + \frac{b}{b-1} + \frac{1}{2}((q'')^2 - q') - \frac{\sqrt[3]{b^5}\sqrt[3]{b^q}}{b-1}, \\ s''_p(q) \geq b^q + c_1 + c_2((q'')^2 - q'')) - c_3\sqrt[3]{b^q}. \end{gathered} \tag{2.51}$$

From the formulas above, (2.50) and (2.51), we can see that the dominant components are b^q, $c_3\sqrt[3]{b^q}$ and $c_3\sqrt[2]{b^q}$. Supposing that q is great enough we can apply the following approximations:

$$s'_p(q) \geq b^q - c_3\sqrt[2]{b^q} = b^q(1 - c_3\frac{1}{\sqrt[2]{b^q}}) \tag{2.52a}$$

$$s''_p(q) \geq b^q - c_3\sqrt[3]{b^q} = b^q(1 - c_3\frac{1}{\sqrt[3]{b^{2q}}}). \tag{2.52b}$$

The result is the proof of the *Theorem (2.4.1)* what I have worded around the formula (2.40). Based on the formulas the increasing q results that the second member tend to zero. Therefore the expression itself asymptotically tend to the value:

$$\lim_{q\to\infty} s^*_p(q) = b^q. \tag{2.53}$$

Formula (2.53) means that, in spite of the high number of *virtual words*, as q increasing the amount of *primary words* get more dominant compared to the *virtual words* considering the length of the words. □

The section related to the asymptotic condition sub-thesis from *Thesis 4.1.2*.

Chapter 3

Interval Merging Binary Tree

In a generic context of an operating support system, more specifically in relationship with the performance management of a telecommunication network, streams of performance counters need to be ingested and transformed towards higher level KPI-s in order to characterize and monitor the performance of such networks. Among several different possibilities a natural choice for such an ingestion layer and real time transformation engine is a stream processor.
Stream processors are cluster level generic processing frameworks allowing real or near real time operations over streams of data. Some of them can be programmed in an SQL-like stream processing language, a construct supporting different processing primitives. Alternatively, and more widely available, it is possible to program such stream processors in high level programming languages such as C++ or Java, like in case of the actually applied Storm framework[6].
Major object-oriented programming languages support data types related to collections, like Java Collection Frameworks[22]. These contain data elements that are related to each other. Implementations range from generic collection types towards specialized ones. One collection type useful for our purposes is SET, which is not allowing the duplication of data elements within. There are three basic operations over SETs: INSERT and DELETE operations modify the number of elements in the SET, while SEARCH operation does not.
SET implementations use special data structures based on the desired trade-off between storage space and the duration of the typical operations defined on sets. When the number of involved elements is well predictable and relatively steady and there is no need for sorted iteration, hash data structure implementations are the best choice. One hash structure that can be used to implement a filter is the Bloom filter [39]. In cases when the number of elements is not well predictable, the wrongly selected capacity may led to too frequent re-hashing which makes the system slower. Additionally, Bloom filters allow for false positives. This implies that the duplication will not only be avoided, but all instances of the data item will be completely filtered out. When sorted iteration is a must or the number of involved elements is not well predictable Binary Search Tree [23] data structure implementations could provide better performance. The original BST was invented in 1960 and serves as the basis of many other advanced variants. B-tree[29] has been proposed as a variant optimizing the movement of large amounts of data. Typical application areas of B-trees are file systems and databases. The main drawback of BST is that the tree may become degenerated after certain sequences of the insert and remove operations. To eliminate the problem of degeneration and make the BST balanced, im-

plicitly improving the performance of the SEARCH operation, AVL Tree [25], [26] has been proposed in 1962. Balancing is time demanding but only when INSERT or REMOVE operations are performed, since these operations may modify the structure of the tree. On the positive side both INSERTION and REMOVE execute embedded SEARCH operations, therefore eventually the operations required to keep the tree balanced do not appear as pure loss compared to unbalanced trees. Red-Black Tree[31],[32] combines the advantages of AVL and B-trees: it is based on a modified B-tree variant and it keeps the tree balanced. (a,b)tree[33] is a balanced tree where all the leaves reside on the same level. There are other trees which are optimized to solve special problems faster or with less resources. Interval tree[34] and Segment tree[37] have been developed to perform a specific search operation on the number of intervals containing a particular key. The reason behind of the so many BST variants is that all of them aiming different trade-off. However the comprehensive comparison of the variants has not been performed until 2004. [38] covers the missing gap: most of the above introduced self-balancing trees are part of the performance analysis. It is common in the above mentioned data structures, except the last two ones, that these trees must store all the keys. In the *Problem* section I will describe why this feature might be a drawback during near real-time duplication filtering from both space and processing time point of view.

3.1 Problem

Given an input stream of keys where the key is a sequence number. Keys are arriving mostly ordered respective to the sequence number. The task is to filter out those entries that arrived already once, meaning that the sequence number has had already this value in an earlier key instance. Additional boundary conditions regarding the arrival pattern apply:

1. upper unbounded range: there is no upper bound of the sequence numbers apart from the limit of the binary representation of this field,
2. lower unbounded range: at any point in time a new key can arrive to the system with a sequence number lower than any sequence number encountered so far,
3. there are long, contiguous intervals of keys with relatively few 'gaps' (missing keys) in between,
4. after a while almost all keys arrive,
5. key duplication (i.e. same key arrived at least twice) on the arrival side is possible due to some reason.

Due to condition 2., a traditional mask cannot be applied for the incoming keys that would drop all keys below a pre-defined value. Thus according to naive approach once all the previously arrived keys have to be stored and the newly arrived ones have to be compared with previously encountered keys.

So as a conclusion, based on the requirements, SET abstraction could be an ideal solution for the problem, however actual implementations suffer from storage space and processing time limitations due to the two-sided unbound range, see Fig.3.1. In this figure

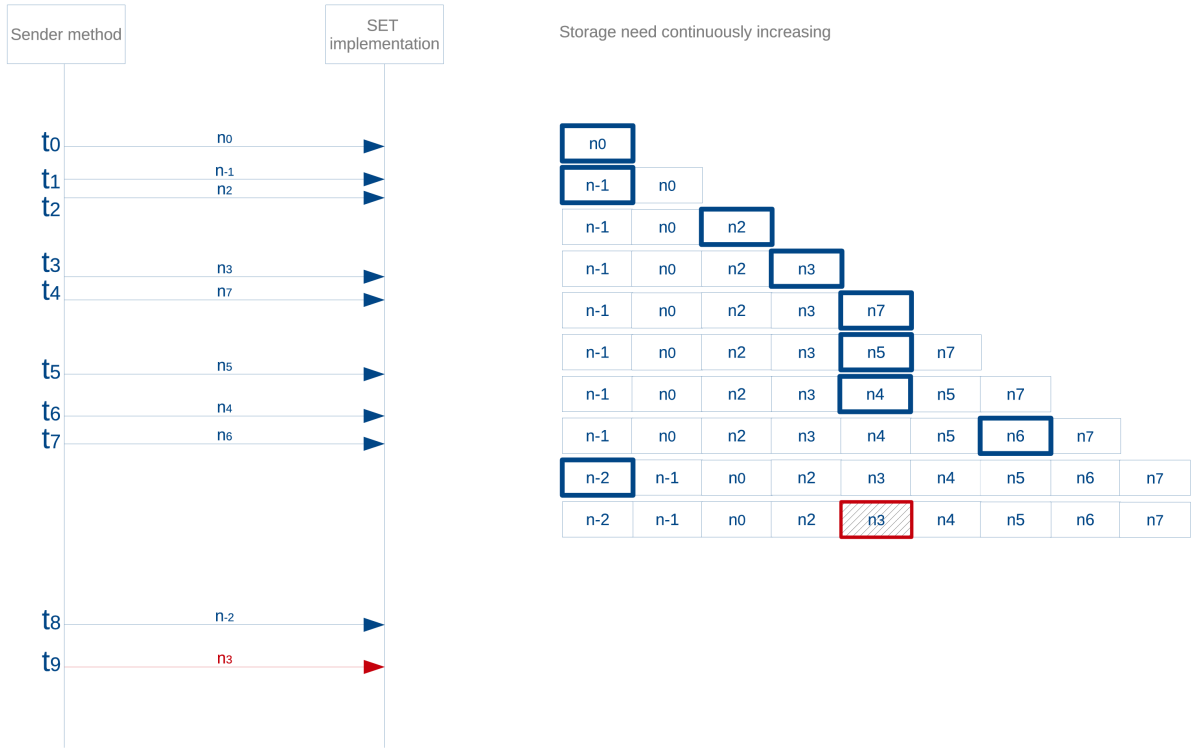

Figure 3.1: Naive approach: the storage need is linearly proportional to all the keys regarding which duplication-free storage should be guaranteed.

timestamps are marked by t_i, while keys are marked by n_j. Dashed arrow and striped rectangle point out to a key, which has already arrived. The storage need is linearly depending on the number of de-duplicated keys, while the search is log_2 proportional if the keys are stored in ordered fashion.

3.2 Methodology

I propose in the subsequent a binary tree that stores intervals of keys rather than individual values, implemented using a JCF SET-like interface. I describe the concept of the tree, then I focus on the insertion operation and the concrete data structure that can optimally implement such a tree.

3.2.1 Concept of the data structure

Let's suppose that keys arrive in the following order:

$$...n_0, n_{-1}, n_2, n_3, n_7, n_5, n_4, n_6, n_{-2}, ...$$

In a naive approach all elements could be stored in a hash or in a binary search tree which is easily searchable, but still the binary search tree or the hash remains an upside-downside open system with infinite storage requirements when keys can arrive with infinite delay. Increments are ordered, only the arrival sequence can be disordered. The first tweak to

the naive approach is to represented arrived keys as pairs. So, elements will be stored like the following:

$$(n_0, n_0), (n_{-1}, n_{-1}), (n_2, n_2), (n_3, n_3), (n_7, n_7), (n_5, n_5),$$

$$(n_4, n_4), (n_6, n_6), (n_{-2}, n_{-2}).$$

At first sight it looks like that we did not win anything, but only doubled the memory footprint. The second tweak is not to automatically put newly arrived elements at the end, but rather to organize the elements in an ordered fashion, filtering at the same time duplicates found during the ordering process. This can be conceptually a sequence of 3 operations: insert at the end, order by key and a filter to skips the entry if it is already found:

$$(n_{-2}, n_{-2}), (n_{-1}, n_{-1}), (n_0, n_0), (n_2, n_2), (n_3, n_3), (n_4, n_4),$$

$$(n_5, n_5), (n_6, n_6), (n_7, n_7).$$

The third tweak is to add an operation that we call interval merging: every pair of neighbor values is checked and if the values are consecutive, the two pairs are converted into one, where the first value of the resulting pair is the first value of the first pair and the second value of the resulting pair is the second value of the second pair. Conceptually the 4^{th} operation can be executed after the order by and filtering operations, but in a more efficient implementation these operations will be covered by a more complex variant dealing with all operations in one INSERT procedure, as described below:

1. The key for ordering is the L-value of an ordered pair and R-value is the second component.

2. If the first key/number arrives store it in the data structure as an ordered pair with the same value stored both as L- and R-values.

3. All successive keys stored according to the following: search the place for the key (remember this is the L-value of the pair) in the data structure

   (a) If it exactly matches with an element in the data structure then drop it $\Rightarrow$ DUPLICATION (message already received)

   (b) If it is the predecessor of the first element check the distance between them (the degree of succession/predecession between them) based on the first element's L-value.

      i. In case the distance is 1 modify the R-value of the generated ordered pair to the R-value of the first element. Then delete the L-value. Insert this pair into the data structure $\Rightarrow$ MERGING.

      ii. In case the distance is higher than 1, simply insert the pair into the data structure.

   (c) If it is a successor of the last element check the distance between them based on the last element's R-value.

      i. If the last element's R-value is smaller than the currently arrived one's R-value with more than 1, insert the currently arrived element into the data structure according to rule 2.

ii. If the distance between the last element's R-value and the currently arrived one is exactly 1, then replace the last element's R-value with the currently arrived one ⇒ MERGING.

iii. If the last element's R-value is higher or equivalent to the currently arrived one's R-value, then drop the actually arrived one ⇒ DUPLICATION (key already received).

(d) If it is in the middle in the data structure both direction checking is required in the following order.

i. Check the predecessor's value.

A. In case the currently arrived element's distance is less than 1, drop the message ⇒ DUPLICATION.

B. In case the currently arrived element's distance is 1 then change the R-value of the predecessor's with the current one's value. Then check the distance from the successor. If it is 1 then change again the predecessor's value (updated with the currently arrived one) to the successor's value ⇒ DOUBLE MERGING.

ii. Else check the distance between the successor's L-value and the currently arrived element's L-value. In case of equivalence change the currently arrived element's value to the successor's L-value. Then delete the successor and insert the newly created pair ⇒ MERGING.

iii. Else insert the newly arrived element into the data structure according to rule 2.

In the following we describe the operation of the algorithm for our small data set:

- n_0 arrives, our data structure will store the following element:

$$(n_0, n_0)$$

- n_{-1} arrives, our data structure will store the following element:

$$(n_{-1}, n_0)$$

- n_2 arrives, our data structure will store the following element:

$$(n_{-1}, n_0), (n_2, n_2)$$

- n_3 arrives, our data structure will store the following element:

$$(n_{-1}, n_0), (n_2, n_3)$$

- n_7 arrives, our data structure will store the following element:

$$(n_{-1}, n_0), (n_2, n_3), (n_7, n_7)$$

- n_5 arrives, our data structure will store the following element:

$$(n_{-1}, n_0), (n_2, n_3), (n_5, n_5), (n_7, n_7)$$

- n_4 arrives, our data structure will store the following element:

$$(n_{-1}, n_0), (n_2, n_4), (n_5, n_5), (n_7, n_7)$$

  Then

$$(n_{-1}, n_0), (n_2, n_5), (n_7, n_7)$$

- n_6 arrives, our data structure will store the following element:

$$(n_{-1}, n_0), (n_2, n_6), (n_7, n_7)$$

  Then

$$(n_{-1}, n_0), (n_2, n_7)$$

- n_{-2} arrives, our data structure will store the following element:

$$(n_{-2}, n_0), (n_2, n_7)$$

So at the end storing only 4 keys, represented as 2 vectors or complex numbers, is required to represent 9 arrived elements. These two complex elements represents two intervals in which all expected keys arrived to the system, this is from where the data structure name origins.
This organization of keys can successfully fulfill the storage complexity related requirements.

3.2.2 Data Structure for Interval Merging

In the previous section an algorithm was introduced that performs an INSERT operation on an imaginary data structure with elements of type value pairs that represent related intervals of ranges of arrived messages without gaps. The data structure can be implemented in many ways impacting the time complexity:

1. in an array element pairs can represent L- and R-values (continuously reserved memory area for a particular type of elements),
2. in a linked list nodes can represent value pairs (not continuously reserved, but references maintained to next/previous nodes as well)
3. in potentially many other ways not listed furthermore,
4. in a newly developed binary tree that we will further elaborate on.

Examining the above mentioned implementations:

1. in case of an array with ordered value pairs the average time complexity is $T(n) = O(log(n))$. However INSERT and REMOVE can require too many movements depending on the place of the affected element in the array. INSERT cause re-indexing of all successor elements and thus moving them one step forward. REMOVE can cause the opposite direction movement.

2. SEARCH in a linked-list is not an efficient operation, however INSERT and REMOVE is very efficient.

As it is visible from the analysis of the first two data structures it seems that a tree-based approach would be effective: in case of a well balanced tree the element-related operations can be quite fast. The question is how to apply element pairs in the nodes. To solve this problem I propose the so-called Interval Merging Binary Tree(IMBT). A node in the tree has the following signature: $\{pointer\_to\_parent, interval\_Left\_Value, interval\_Right\_Value, pointer\_to\_Left\_Child, pointer\_to\_Right\_Child\}$.
NULL parent pointer means that this particular element is the root node of the data structure. NULL $pointer\_to\_Left\_Child$ and $pointer\_to\_Right\_Child$ means leaf node just as in a traditional binary search tree.
A node always stores value pairs, which are the $interval\_Left\_Value$ and the $interval\_Right\_Value$, despite elements to be inserted, removed, searched for being single keys. The relation between a parent node and a $Right\_Child$ of the parent node is that the $interval\_Left\_Value$ of the child node must be higher by two or more than the $interval\_Right\_Value$ of the parent node. The relation between a parent node and a $Left\_Child$ of the parent node is that the $interval\_Right\_Value$ of the child node must be smaller by two or more than the $interval\_Left\_Value$ of the parent node.
Additionally, pointers to parents and left or right elements are substituted by single lines. The left and right values must be indicated. Fig.3.2 is the visualization of the example from previous section, based on the graphical representation of IMBT and the INSERT concept:

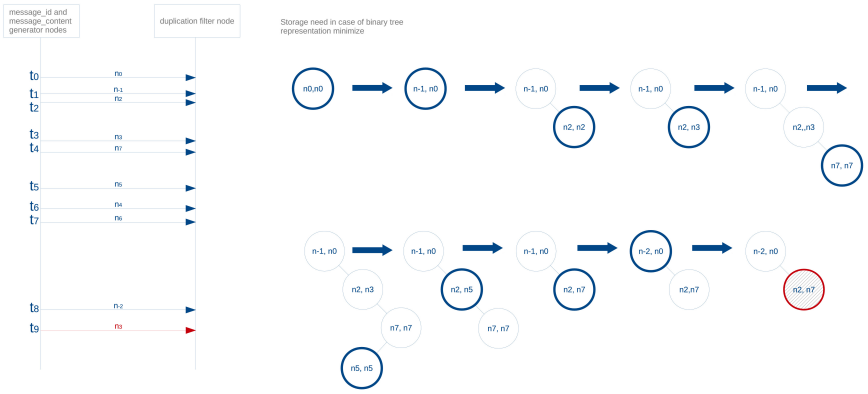

Figure 3.2: The evolution in time of the IMBT based representation

The section relates to *Thesis 4.2.1*.

3.3 State-space analysis

.

In contrast to a completely balanced binary search tree, it is impossible to associate to the newly developed data structure a one dimensional (the number of input key dependent) function to determine for instance the average cost of an operation.

Nevertheless, for further development, it is essential in case of any data structure to determine the actual boundaries of their applications.

In this section I introduce the hardness of the identification of the state space of this data structure, which is needed for the later performance analysis and other input pattern based classifications. During the modeling Fibonacci sequences, bipartite multi-graphs and combination tables are applied.

3.3.1 Inteval State-space

The surprising thing regarding the scenarios visible at Fig.3.3, Fig.3.4 and Fig.3.5 is that the input size is four in all cases.
As it is visible in case of four keys ($N = 4$), based on the possible number of neighbors,

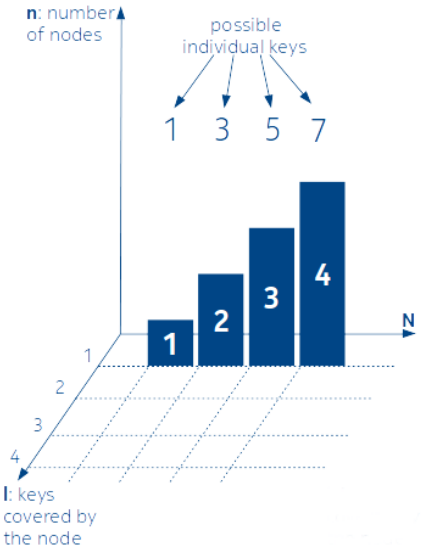

Figure 3.3: IMBT interval evolving when no direct neighbor exists.
In the figure N represents the T time as well. By looking to the figure from the right side, the remaining axes display a histogram of the intervals in different moments.

the following scenarios can be distinguished:

- None of the keys are neighbour of each other, like Fig.3.3,
- Two of them are neighbours and the other two are not,
- Two of them are neighbours and the remaining ones as well,
- Three of them are neighbour and one is not, like Fig.3.5,
- All the keys are neighbour of each other, like Fig.3.4.

Therefore I can say that according to Hardy and Ramanujan [56]:

Theorem 3.3.1. the number of possible interval states in case of IMBT, at $T = N$ time, is equal with the number of ways N can be written as a sum of positive integers:

$$\lim_{N\to\infty} p(N) \approx \frac{1}{4N\sqrt{3}} e^{\pi\sqrt{2N/3}} \tag{3.1}$$

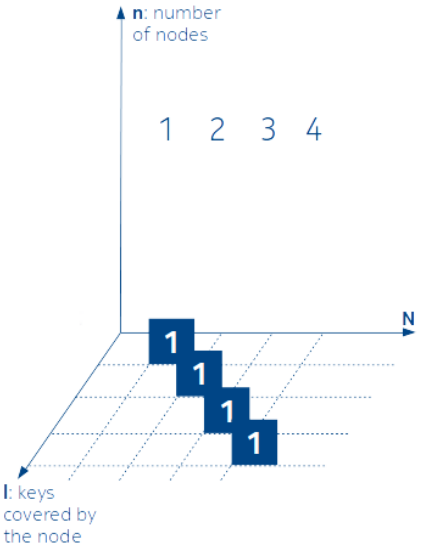

Figure 3.4: IMBT interval evolving when the keys are subsequent

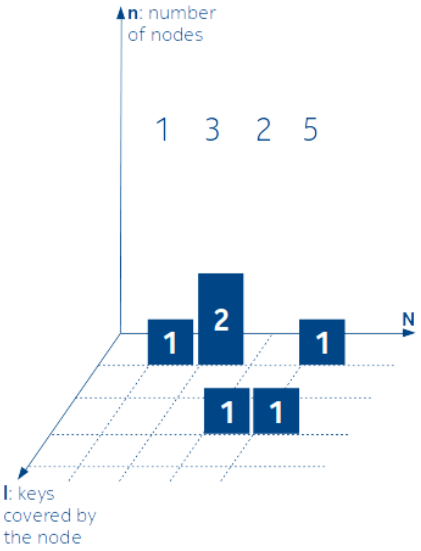

Figure 3.5: IMBT interval evolving when there are both neighbour and stand alone keys

We can identify the addends of the sum as the individual interval lengths of the nodes in the IMBT. In this case for the a average interval lengths, considering the list items above, we get the following values, respectively: $4/1 = 4$, $4/2 = 2$, $4/2 = 2$, $4/3$, $4/4 = 1$. As it is visible there are two equivalent values: 2. Therefore it is generally true that the number of integer partitions is a rough upper estimation regarding the possible number different averages for a given input size N.
Additionally $p(N)$ does not say anything about the weight of the intervals based on their position in the tree. Since the same decomposition may led to very differently weighted arrangements it is matter if for instance the intervals of 8 is written in e.g..: $1+1+4+1+1$ or $4+1+1+1+1$ or $1+1+1+1+4$ form. Supposing that the intervals are organized into a balanced binary search tree, the cost of the search operation in first case is the most favourable, and the last one is the least favourable case.
Therefore in *Traversal Strategy Based Weight Classes* I will examine IMBT from another point of view.

3.3.2 Traversal Strategy Based Weight Classes

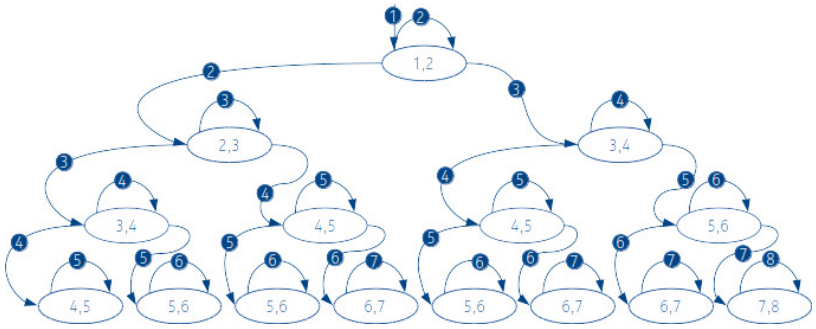

Figure 3.6: IMBT weight classes caused by the traversal strategy

In Fig.3.6 the arrows with number represent the j^{th} comparison during the SEARCH operations. Here I would like to mention that, for the sake of simplicity, during the comparisons the less or equal will be considered as one atomic step. The dark background of the number expresses that the result of the comparison can be positive (that is, the node cover more than one keys). In this figure, instead of boundaries of the intervals, the same information is displayed in the nodes like on the arrows as well.
As we can see if the key to be searched for is equal with the left hand value of the root node then exactly one comparison will be performed. If the key to be searched for is in between the left and right hand values of the root node or equal with the right hand value then two comparison will be performed.
If the key is greater than the right hand value of the root node and fall into the interval of the right hand child's left and right hand values then three or four comparisons are required, depending on the exact value.
If the key is less than the left hand value of the root node and fall into the interval of the right hand child's left and right hand values then two or three comparisons are required, depending on the exact value.
By continuing the examination of the distribution of the different classes of intervals, based on the required number comparisons, we can recognize the following rules (in case of the intervals are organized into a completely balanced tree).
Considering the root (first) level there is one from that interval where $(1, 2)$ comparison can occur. At the second level there is one interval where $(2, 3)$ and one interval where $(3, 4)$ comparison(s) can occur. Finally, at the third level the cumulated number of intervals where $(1, 2)$ and $(2, 3)$ comparison(s) can occur is unchanged. However, the number of $(3, 4)$ comparison based intervals is increasing from one to two. Additionally two $(4, 5)$ and one $(5, 6)$ comparison need interval appears.
By the cumulative number of types, as more and more layers are taken into account, we will get the pattern visible at Table-1. Examining carefully the lists we can realize that

Theorem 3.3.2. the central element of each row composed from cumulative number of weight types is the Fibonacci sequence itself. The numbers in the lists (lines in this case), preceding the central elements, are also the evolving Fibonacci sequences themselves. The rest of the numbers must satisfy the requirement that the sum of the numbers is equal with $2^n - 1$ in every n^{th} line.
However, another rule also can be recognized there:

Theorem 3.3.3. the numbers in a line from Table-3.3.2 are composed as the sum of the two preceding numbers of the previous line, except the last one which is always zero.

Table 3.1: Distribution of weight classes in case of the IMBT is completely balanced. The Fig.3.6 snapshot is marked with bold.

Total number of nodes in IMBT	Distance from the root [number of comparisons regarding the left hand value]										
	1	2	3	4	5	6	7	8	9	10	11
1	1										
3	1	1	1								
7	1	1	2	2	1						
15	**1**	**1**	**2**	**3**	**4**	**3**	**1**				
31	1	1	2	3	5	7	7	4	1		
63	1	1	2	3	5	8	12	14	11	5	1

Table 3.2: Fibonacci sequences in the cumulated weight classes

1										
1	**1**	1								
1	1	**2**	2	1						
1	1	2	**3**	4	3	1				
1	1	2	3	**5**	7	7	4	1		
1	1	2	3	5	**8**	12	14	11	5	1

Until now I have shown that there are two distinct components during the examination of IMBT state space. One is if how many ways the number of keys can be decomposed into integer partitions.
Another component of the state space (the weight classes) is based on the number of nodes and depends on the associated traversal strategy.
Now, to be able to determine the combined number of input pattern classes somehow we have to put these components together. In *Bipartite Graphs and Combination Tables on the modeling of IMBT State Space* I will present this combination procedure and the resulting mathematical models.

3.3.3 Bipartite Graphs and Combination Tables on the modeling of IMBT State Space

To be able to start the combined analysis we will perform the following mappings.
Let's denoting the length of the interval belongs to an n_i node from the IMBT by $l_i \in L$, where L is a multi-set. Then we map the set of same length of intervals onto $i_1, i_2, ..., i_k \in I$ elements. This means that by having the $L = \{l_1, l_2, ..., l_n\}$ lengths, where the values of $l_h = l_i = ... = l_j$ is equal, then this fact results one new element, i_p, in the I set. That is the following $l_h \rightarrow i_p, l_i \rightarrow i_p, l_j \rightarrow i_p$ surjection is performed in case of $l_h = l_i = l_j$. Therefore $k \leq n$.
Let's denoting the number of comparison required to achieve the left hand value of an arbitrary n_i node by $s_i \in S$, where S is a multi-set. Then let's map the traversal strategy based identical comparison weight types onto $w_1, w_2, ..., w_j \in W$ elements. This means that by having the $S = \{s_1, s_2, ..., s_n\}$ lengths, where the values of $s_h = s_i = ... = s_j$ is

equal, then this fact results one new element, w_p, in the W set. That is, the following $s_h \to w_p, s_i \to w_p, s_j \to w_p$ surjection is performed in case of $s_h = s_i = s_j$. Therefore $k \leq n$.
Since the newly defined I and W are two disjoint sets we can consider them as the vertices of a $G(I, W)$ bipartite (multi-)graph. We will assign degrees to each vertex according to the following manner:
The degree of each i_i vertex is equivalent with the number of that particular interval lengths. According to this in case of $l_h = l_i = l_j$ the degree of the associated i_i vertex is $d(i_i) = 3$.
The degree of each w_i vertex is equivalent with the number of that particular weight types in the search tree.
Therefore we can write that

Theorem 3.3.4. $\sum_{i=1}^{j} d(w_i) = \sum_{i=1}^{k} d(i_i) = n = |E|$, where $E = \{e_1, ..., e_n\}$ is the set of the e_i edges of $G(I, W)$. That fact that the above two sets, I and W, are the independently different classifications of the same nodes of the IMBT implies that he sum of the degrees of the vertices in both sets are equivalent with n. □ Considering an IMBT arrangement/configuration where $n = 4$, and both I and W sets contain one-one vertex with degree two, and two additional vertices with degree one-one. So, $d(i_1) = d(w_1) = 2$ and $d(i_2) = d(i_3) = d(w_2) = d(w_3) = 1$. At this moment regarding N we can only say that $N \geq n$.
It is obvious that to get the above I set two of the lengths must be equal, eg. $l_1 = l_2$, and the other must differ from both $l_1 = l_2 \neq l_3$, $l_1 = l_2 \neq l_4$ and $l_3 \neq l_4$. *Definition:* Those L interval length multi-sets are called *interval lengths ratio base classes*, denoted by L^b, in which

- at least one l_i exists which is co-prime to all the other l_j, such that $i \neq j$ supposing that $l_i \neq l_j$, or
- if $l_i = l_j$ for all $i \neq j$, than $l_i = l_j = ... = l_k = \textit{prime number}$.

That is, L is an L^b if

$$\exists l_i \in L \mid (\forall i \neq j \wedge l_i \neq l_j \Rightarrow gcd(l_i, l_j) = 1) \vee (\forall i \neq j \ \Rightarrow l_i = l_j = prime\_number). \quad (3.2)$$

Therefore, if eg.: $L = \{l_1, l_2, l_3, l_4\}$ is an interval lengths ratio base class, that is $L = L^b$, then L^b determines all the $N_1, N_2, ...$, which differ from each other by only an integer multiplication for a given $(L^b, n = |L^b|)$ pair. This representation/decomposition is unique, except for the order of the factors:

$$\begin{aligned} N_x &= (d(i_1) \times l_1 \times x) \ + \ (d(i_2) \times l_3 \times x) \ + \ (d(i_3) \times l_4 \times x) = \\ &= x \times (d(i_1) \times l_1 \ + \ d(i_2) \times l_3 \ + \ d(i_3) \times l_4) \end{aligned} \quad (3.3)$$

where $x \in \{1, 2, 3, ...\}$. If n is given that is the maximum information we can get, regarding N. In Fig.3.7 we can see all the different possible configuration for the above $G(I, W)$, where $|I| = |W| = 3$ and $|E| = n = 4$. That is, there are three-three vertices on both sides of the G graph. About N we can say that $N \geq 4$. If we are aware of the $l_1, l_2, l_3, l_4 \in L$ values, e.g.: $l_1 = l_2 = 1, l_3 = 2$ and $l_4 = 3$ and therefore $L_1 = L^b$ then we can say that $N_1 = 7$. However, $N_2 = 14, N_3 = 21, ...$ and $L_2, L_3 \neq L^b$. As it is visible from the Fig.3.7 there are seven different possible configuration exists. Regarding the number of possible configurations, in case of a given (L, n) pair, till now we have got a mathematical model as a $G(I, W)$ bipartite graph. We state that

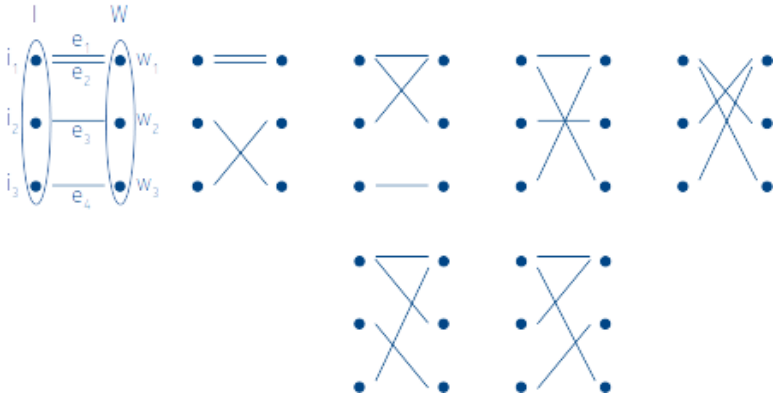

Figure 3.7: G(I,W), where $|I| = |W| = 3$ and $n = 4$

Theorem 3.3.5. the simplified adjacency matrix representation of a $G(I, W)$, which is derived from an IMBT according to the above process, can be corresponded to a contingency table. Given an $G(I, W)$ bipartite graph derived from an IMBT. Let's prepare the adjacency matrix of $G(I, W)$, where parallel edges are allowed, on the following manner. Since $G(I, W)$ is a bipartite graph there is no edges between the vertices belong to the same vertex set. Then we will apply the following simplification: instead of enumerate all the points from both sets on the right side and the top of the adjacency matrix merely the points from I will be displayed with the associated $d(i_i)$ values on the right side. On the top of the matrix only the points from W will be displayed with the associated $d(w_j)$ and values.
The edges appear in the matrix as numerical entries in the cells of the matrix. The value of a particular cell represents the number of edges between the i_i and w_j points. However the $d(i_i)$ and $d(w_j)$ values are constraints about the sum of a given i row and j column. From *Theorem 3.3.4* we know that the sum of cells in a row is equivalent with the degree of that particular vertex. The same is true for all column. Therefore the sum of sums of every row is equivalent with the sum of sums of every column. This feature of the simplified adjacency matrix is corresponding to a contingency or combination table, which may contain discrete samples about the same multitude from two different point of view. □
At Fig.3.8 the simplified adjacency matrix representation of the $G(I, W)$ graphs from the Fig.3.7 are visible. Since the edges do not appear directly, the simplified adjacency matrix

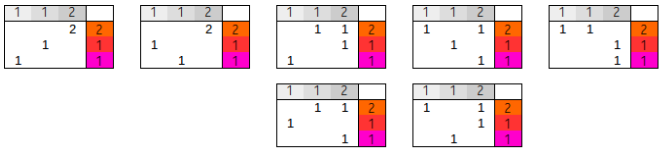

Figure 3.8: Simplified adjacency matrix of G(I,W)

remains unchanged in case of two different, an e_k and e_l edges, which are not sharing on any vertices on none of their ends, are mutually replaced with each other. It is true for that case as well when neighbours edges, sharing on a multi-degree vertex, replacing their non-sharing ends with each other.
Therefore from the simplified adjacency matrix like this it is still hard to say at least the formal condition when we may speak about different states, that is the total weight of the IMBT. The number of states for a given (I,W) is the different number of total weights of the IMBT. Nevertheless, we can apply the following transformation without violate the

validity of the transformed model. During the transformation we are composing so called domains in the matrix on that way that every row(or column) with value $d(i_i)$ (or $d(w_j)$) will be substituted with $d(i_i)$ (or $d(w_j)$) rows(or columns), where the constrain value of each row is '1'. Therefore the 1×1 cells, which are in the cross of the $d(i_i)$ row and the $d(w_j)$ column, will be replaced by such a domain which consists of $d(i_i) \times d(w_j)$ cells.
In Fig.3.9 the domain composition of the above $G(I, W)$ is visible, where the domains are marked/surrounded by dotted lines.

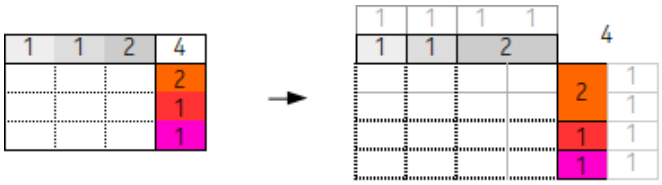

Figure 3.9: G(I,W) simplified adjacency matrix transformation to domain representation

In Fig.3.10 the domain transformed matrix representation of the Fig.3.7 examples are visible. The numbers with blue background mark the related $G(I, W)$ from the examples.

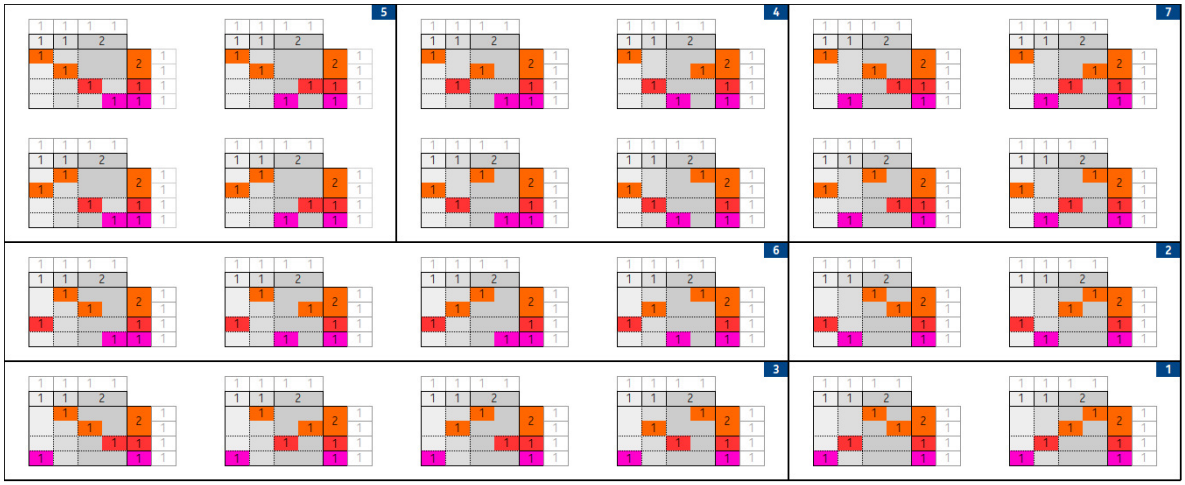

Figure 3.10: G(I,W) examples with domain representation.

Let's denote the set of all the $G(I, W)$ graphs belong to the same partition of N by P_{N,L_i}. From the *Interval State Space Section* we know that $i \in \{1...p(N)\}$. A particular $G_k(I, W) \in P_{N,L_i}$ expresses the n members sum of two members products, where the members of the products are from the L_i and W sets respectively. Therefore the $G_k(I, W) \in P_{N,L_i}$ determined sum of products can be mapped onto the IMBT state space. Now I will define the subset of P_{N,L_i}, denoted by P^s_{N,L_i}, according to the following. P^s_{N,L_i} is that subset of the P_{N,L_i} set, which contains the maximum number of $G_i(I, W)$ graphs from P_{N,L_i}, so that in the $G(I, W)$ associated transformed matrices regarding the sum of cells at least in four domains are different for all the (G_i, G_j) $i \neq j$ pairs.

Theorem 3.3.6. $|P^s_{N,L_i}|$ is an upper bound regarding the possible number of IMBT states belongs to an $N \rightarrow L_i$ partition. Considering the following $l_1, l_2, ...l_n$ lengths and the $s_1, s_2, ...s_n$ steps. Supposing that there are i elements from both the l's and s's where the associated lengths and steps are equivalent with each other. Additionally there are two additional j and k number of elements from both L and S within where the associated

values are the same and $i + j + k \leq n$. Let's the associated value of the i elements are $v_i = 2$, $v_j = 3$ and $v_k = 4$. Then there will be such a $G_1(I, W)$ and $G_2(I, W)$ bipartite graphs, which identical in every other pairings regarding the member of the products except the $G_1 \to r_1 = ... + l_{i,i} \times s_{i,i} + l_{j,1} \times s_{j,1} + ... + l_{j,j} \times s_{k,1} + l_{k,1} \times s_{j,j}$ and $G_2 \to r_2 = ... + l_{j,1} \times s_{i,i} + l_{i,i} \times s_{j,1} + ... + l_{k,1} \times s_{j,j} + l_{j,j} \times s_{k,1}$. In this case (G_1, G_1) pair satisfies the above condition regarding the sum of domains, however the associated r_1 and r_1 results are identical, therefore represents the same state of IMBT. □ Now we can word

Theorem 3.3.7. the upper bound of IMBT state space in case of merely N is given and the same n number of lengths are always sorted into the same tree structure no matter whatever it is:

$$IMBT_{States}(N) = |P^s_{N,L_1} \cup P^s_{N,L_2} \cup ... \cup P^s_{N,L_{p(N)}}| \leq \sum_{i=1}^{p(N)} |P^s_{N,L_i}|. \tag{3.4}$$

In case of the IMBT is completely balanced then the degrees belongs to a particular w_i is equivalent with the corresponding number from the corresponding line of Table-3.3.2. For instance in case of $n = 7$ we can identify the third line of Table-3.3.2. Therefore we know that the number of different weights are 5. And the seven nodes are sorted into five classes according to the followings $d(w_1) = 1, d(w_2) = 1, d(w_3) = 2, d(w_4) = 2, d(w_5) = 1$.

The section relates to *Thesis 4.2.2.*

3.4 Arrangements Related Conditions, Theorems, and Equations

By now we know a model in which we can count the average cost of search operation by simple multiplication of the corresponding values in the contingency table. In the following, I will define some metrics and through these we will examine the evaluation of the contingency table.
Let us denote by N the number of keys stored in the IMBT so far just like above. However, unlike above, to be able to examine the evolution of the number of nodes in the tree, we introduce a random variable, V_i, which is the instantaneous number of vertices (nodes previously) in the tree at time instant i, where $i \in 1...N$. It is obvious from the definition that $V_1 = 1$ and V_i can be mapped to states in a stochastic matrix. The distribution of the lengths of the intervals is affected by the homogeneity and the finite/infinite nature of the stochastic matrix.
I define the series of instantaneous average interval lengths by the following formula:

$$\bar{L}_i = \frac{l^i_1 + l^i_2 + ... + l^i_{V_i}}{V_i}, \tag{3.5}$$

where i is the time instant ($i \in 1...N$) and l^i_k is the length of the interval stored by node k at time instant i. $\bar{L}_i$ is a random variable as well. We assume that for the series of $\bar{L}_i$ the following constraints are true:

- There is an expected value a, to which the individual random variables, a_i, stochastically converge to as N tends to infinity, where $a = \lim_{N\to\infty} (a_1 + a_2 + ... + a_N)/N$.

- There is a $c = (\sigma_1^2 + \sigma_2^2 + ... + \sigma_N^2)/n$ independently from N, where σ_i is the standard deviation of the interval lengths at time instant i.
- There is a non-negative function $r(x)$ for which $r(0) = 1, \lim N \to \infty (r(1) + r(2) + ... + r(n))/n = 0$, and additionally $|corr(\bar{L}_i, \bar{L}_i)| \leq r(|i - j|)$, $i, j \geq 1$.

The conditions mentioned above together constitute the Bernstein-theorem [64]. According to the theorem, if the three constraints are simultaneously met, then the weak law of large numbers is true.
Using the Bernstein theorem as a starting point, we can identify two types of completely different input pattern classes, for which the behavior of contingency tables is examined and the cost of average search operation is determined:

- In the first case only a nonrecurring, transient, infinite state stochastic matrix can be composed based on the associated states V_i. Additionally we assume that the Bernstein-theorem is true for the series of $\bar{L}_i$,
- In the second case, we assume that based on the state V_i, it is possible to create a stochastic matrix which has a finite state-space, is aperiodic, irreducible (that is ergodic), and recurrent.

The satisfaction of the first criterion implies that the average interval length is upper bounded. As a result, the variance of the interval lengths is also upper bounded, therefore, the scenario in which we have a composition of a large interval with continuously increasing length with increasing number of small ones, is not valid. Rather, as N increases, there will be an increasing number of gaps between the intervals, associated with permanently missing keys, that is, keys where the probability of arrival of the associated packet converges to zero. Both from the previous fact and from Equation (3.5) it is obvious that V_i is proportionally increasing as well. The satisfaction of the second criterion implies the presence of temporary gaps only: in spite of the increasing N the finite number of nodes implies that the instantaneous average interval length is increasing, that is the number of gaps is upper bounded. *During the deductions, it is assumed that any key has the same probability to be searched for. That is $P(the\ key\ we\ are\ looking\ for\ is\ k_m) = \frac{1}{N}, where\ m \in (1...N)$.

3.4.1 Permanent Gaps

In the case of permanent gaps, the mean of the average interval length is a constant value, a. We do not exclude the possibility of having temporary gaps, due to the out-of-order arrival of packets. As a result, the header $d(i_i)$ in the associated contingency table will follow a kind of distribution. Taking into account the effect of temporary gaps in the analysis would make the analysis more complex, but their effect is minor, therefore they will be discarded in the subsequent.

Linked List Arrangement

In this realization, our additional assumption against the keys is that there is a smallest one. The tree degenerated into a linked list and three associated contingency tables with $V_i = n = 3$, $V_i = n = 7$ and $V_i = n = 15$ are shown in Figure 3.11.

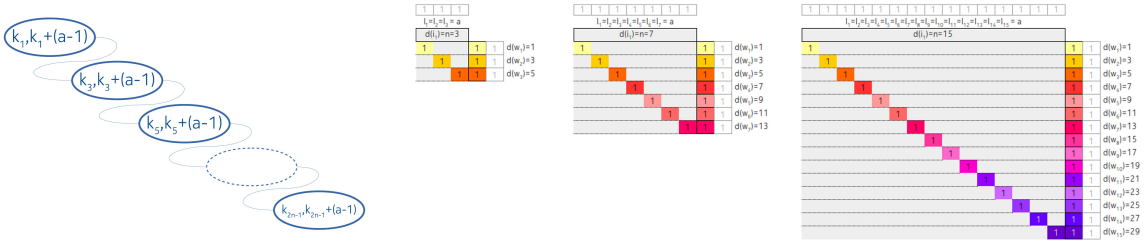

Figure 3.11: Linked list degenerated IMBT and three associated contingency tables.

Theorem 3.4.1. In the case when there is no shuffling and no balancing at all, the tree degenerates to a linked list and

$$A(N,a) = \frac{N}{a} + \frac{a-1}{a}. \tag{3.6}$$

From the contingency table, it is clearly visible that the w_i follows the sequence of odd numbers and $d(w_i)$ remains constant. **Proof of *Theorem 3.4.1***: Every node contains two keys, since a node covers an interval and the keys represent the borders of that particular interval. Let us assume that a node in the tree covers a keys on average. Then we can denote by k_i the starting key and by $k_i + (a-1)$ the ending key. Additionally, due to the non-overlapping feature of IMBT, it is also trivial that $(k_i + a - 1) < k_{i+1}$. First let us see the $a = 1$ case. In this case in the linked-list degenerated data structure the starting and the ending keys are equal. As a consequence: if the key to be searched for is not equal with k_i then the second comparison with the right value of that particular node is necessary but, due to $k_i == k_i + a - 1$, the outcome of the comparison is always false. Since the number of the nodes is $n = N/a$, and here $a = 1$, therefore $n = N$. Based on this, we can write that the average cost of SEARCH is equal to the expected value:

$$\begin{aligned} A(N) &= \frac{1}{N}\sum_{i=1}^{n}(2i-1) = \\ &= \frac{1}{N}\left[2\frac{n(n+1)}{2} - n\right] = N. \end{aligned} \tag{3.7}$$

Now, we examine the $a \geq 2$ case. It is still valid that $n = N/a$. Analogous to the previous deduction we can write that the average cost of the SEARCH operation is:

$$\begin{aligned} A(N,a) &= \frac{1}{N}\sum_{i=1}^{n}(2i-1) + 2i(a-1) = \\ &= \frac{1}{N}\Big[n(n+1) - n + (a-1)n(n+1)\Big] = \\ &= \frac{N+a-1}{a} = \\ &= \frac{N}{a} + \frac{a-1}{a}. \end{aligned} \tag{3.8}$$

It is visible that substituting 1 into a we will return Equation (3.7). An additional consequence is that as a grows the equation tends to:

$$\frac{N}{a} + \frac{a-1}{a} \approx \frac{N}{a} + 1. \tag{3.9}$$

With the deduction above, Theorem 3.4.1 is proved. □

Completely Balanced Arrangement

We assume a completely balanced tree with the number of node power of 2, that is $n = 2^l - 1$. Three associated contingency tables with $V_i = n = 3$, $V_i = n = 7$ and $V_i = n = 15$ are shown in Figure 3.12.

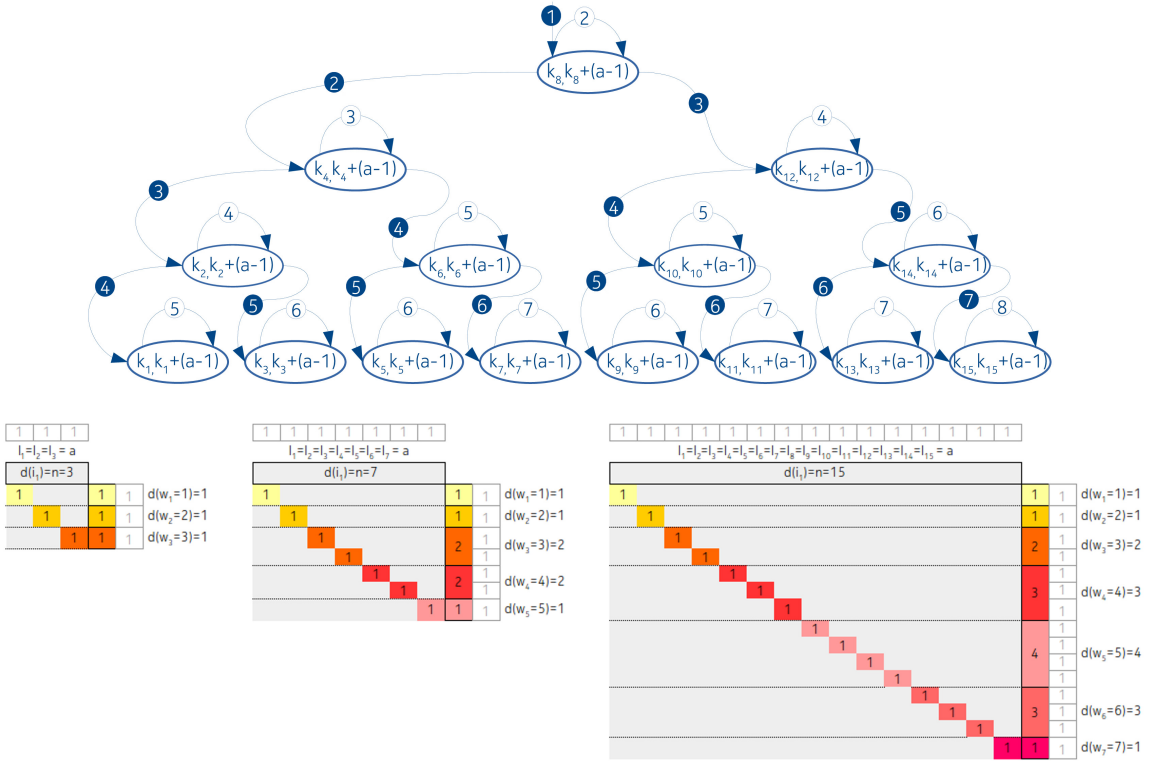

Figure 3.12: Completely balanced IMBT and three associated contingency tables.

Theorem 3.4.2. With the above conditions, the average cost of the search operation can be expressed with the following formula:

$$A(N,a) = \frac{3}{2} log_2\Big(\frac{N}{a} + 1\Big) + \frac{3a}{2N} log_2\Big(\frac{N}{a} + 1\Big) - \frac{a+1}{a}. \tag{3.10}$$

As is visible from the figure and as we have proven in 3.3.2, the balancing has a typical fingerprint in the $d(w_i)$ distribution: It follows the Fibonacci sequence until the middle of the rows on the way from the top rows to the bottom rows. Supposing that $N >> 0$, and $a > 0$. Then we can apply the following simplification on Equation (3.10):

$$A(N,a) \approx \frac{3}{2}\log_2(\frac{N}{a}) = \frac{3}{2}\log_2(N) - \frac{3}{2}\log_2(a) = \tag{3.11a}$$

$$= \log_2(N) + \frac{1}{2}\log_2(N) - \frac{3}{2}\log_2(a). \tag{3.11b}$$

That is, an IMBT with the given criteria will outperform a BST as long as the difference of the second and third term is negative, in other words, provided that:

$$a > \sqrt[3]{N} \tag{3.12}$$

From now on, the number of layers or levels is denoted by l (in contrast to the previous notation of lengths). The root node is the $l = 1$.

Proof of Theorem 3.4.2: Based on our notations we can write that:

$$\begin{aligned} n &= \frac{N}{a}, \\ l &= log_2(n+1) = log_2(\frac{N}{a}+1). \end{aligned} \tag{3.13}$$

During the deduction we will use the following identity:

$$\sum_{i=1}^{n} i2^i = n(2^{n+1}-2) - (2^{n+1}-4) + (n-1)2. \tag{3.14}$$

Based on the relations (Equation (3.13)) we can define the layer level sums:

$$2^{i-1} + 2^i(a-1) + (i-1)2^{i-2}3 + (i-1)2^{i-2}3(a-1), \tag{3.15}$$

where i means the i^{th} level in the tree. However, this form is not suitable for equal transformations. Therefore, we split the formula into a fixed member which is the first node and the layer level members. Due to this split we will use an incremented i and the counter in the sum will last to $l-1$ instead of l:

$$\begin{aligned} A(N,a) =& \frac{1}{N}\Big[1 + 2(a-1) + \sum_{i}^{l-1} 2^i + (a-1)2^{i+1} + 3i2^{i-1} + 3(a-1)i2^{i-1}\Big] = \\ =& \frac{1}{N}\Big[1 + 2(a-1) + \sum_{i}^{l-1} 2a2^i - 2^i + \frac{3a}{2}i2^i\Big] = \\ =& \frac{1}{N}\Big[1 + a - a2^l - 2^l + \frac{3a}{2}l2^l\Big]. \end{aligned} \tag{3.16}$$

Since

$$\begin{aligned} a - a2^l =& -a(2^l - 1) = -N \\ 1 - 2^l =& -\frac{N}{a} \end{aligned} \tag{3.17}$$

we can write that

$$\begin{aligned} A(N,a) =& \frac{1}{N}\Big[\frac{3a}{2}l2^l - N - \frac{N}{a}\Big] = \\ =& \frac{3a}{2N}(\frac{N}{a}+1)log_2(\frac{N}{a}+1) - 1 - \frac{1}{a} = \\ =& \frac{3}{2}log_2\Big(\frac{N}{a}+1\Big) + \frac{3a}{2N}log_2\Big(\frac{N}{a}+1\Big) - \frac{a+1}{a}. \end{aligned} \tag{3.18}$$

With the deductions above, Theorem 3.4.2 is proved. □

3.4.2 Temporary Gaps

Let us assume in the following cases that it is possible to compose from V_i a finite state $\{1...p\}$, ergodic (aperiodic, positive recurrent), at least partially recurring stochastic matrix. Let us denote by M_j the mean recurrence time in state j. If M_j is finite then state

j is positive recurrent. The state j in which the expected return time is the smallest one, will represent the number of nodes in the IMBT as a steady-state value. Therefore we can substitute that state with value $n = j$. Regarding the distribution of the interval lengths, which are necessarily increasing, we will distinguish two possible realizations.

- In the first realization we suppose that the increasing interval lengths are uniformly distributed to nodes, the number of which is fixed.
- In the second realization the distribution is not uniform.

Linked List Arrangement

In this subsection, our additional assumption against the keys is that there is a smallest one, which we may always call first and as time goes by the probability that all the keys near the first key have already arrived is increasing. Since the gaps are temporary ones, out-of-order arrival implies there are keys which arrive late, therefore temporary side-branches might appear over time, outside of the main branch. The length of these temporary branches depends on the statistics of the out-of-order arrival pattern. Subsequent side-branches are not taken into account because their effect is marginal.

Theorem 3.4.3. With the distribution characterized above, the data structure degenerates into a linked list. Suppose that the increasing lengths are uniformly distributed across the nodes. Then, due to the uniformly distributed increasing lengths, the average cost of the search operation does not depend on N:

$$A(N, a) = n. \tag{3.19}$$

Figure 3.11 accurately describes this scenario as well.
In the following, we suppose that the distribution of the lengths is not uniform, but node-heavy, meaning that every node contains a single key only, except one, which contains all the remaining $N - n + 1$ keys. That heavy node can reside at the tail, in the middle, or at the root of the linked list degenerated tree.

Theorem 3.4.4. Then the average costs of search operations in case of node-heavy arrangements are the followings:

$$\lim_{N\to\infty} A_{tail\ heavy}(N, a) = 2n. \tag{3.20a}$$

$$\lim_{N\to\infty} A_{middle\ heavy}(N, a) = n. \tag{3.20b}$$

$$\lim_{N\to\infty} A_{root\ heavy}(N, a) = 2. \tag{3.20c}$$

The related arrangements and contingency tables are shown at Figure 3.13 and Figure 3.14 respectively. The proof of Theorem 3.4.3 is easily derivable from Equation (3.9), with the substitution of $n = \frac{N}{a}$:

$$A(N, a) = \frac{N}{a} + \frac{a-1}{a} \approx n + 1 = C. \ \square \tag{3.21}$$

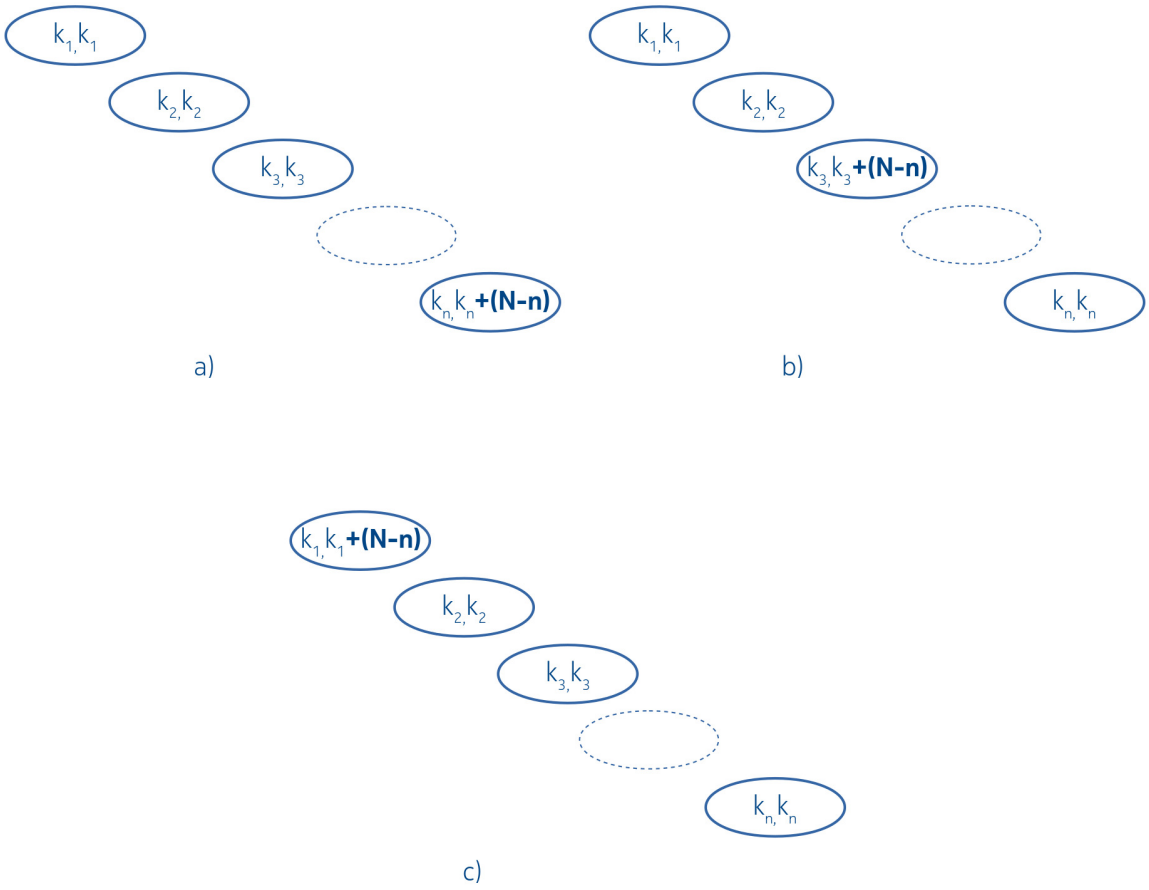

Figure 3.13: The linked list degenerated IMBT with heavy nodes.

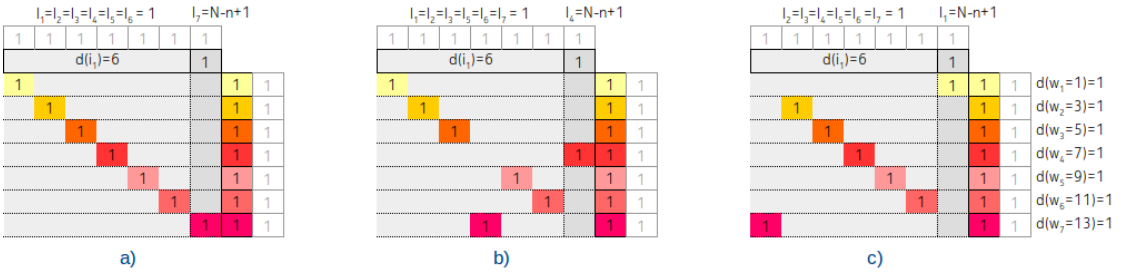

Figure 3.14: The associated contingency tables of linked list degenerated IMBT with heavy nodes.

The proofs of node heavy cases, Theorem 3.4.4, are the following. According to arrangement *a)* we can write that $n = N/a$, and

$$A(N,a) = \sum_{i=1}^{n} \frac{2i-1}{N} + 2n\frac{N-n}{N} = \\ = N\left[\frac{2a-1}{a^2}\right] = n\frac{2a-1}{a}. \tag{3.22}$$

It is visible from the results that as N and a grow along with $n = const.$ the value tends to $2n$, which is not surprising considering that accumulation is possible merely in the last

element. Considering arrangement *b)* we will get that

$$A(N,a) = \sum_{i=1}^{n} \frac{2i-1}{N} + \frac{n+1}{2}\frac{N-n}{N} = \\ = \frac{1}{2}\frac{N+a-1}{a}. \tag{3.23}$$

Comparing the result to Equation (3.21) we can see that the average SEARCH cost is half of that of the uniformly distributed one. Arrangement *c)* can be expressed by the following equation:

$$\begin{aligned} A(N,a) &= 2\frac{N-n}{N}\sum_{i=1}^{n}\frac{2i-1}{N} = \\ &= 2\frac{N-n}{N} + \frac{2}{N}\sum_{i=1}^{n} i - \frac{(n-1)}{N} = \\ &= 2\frac{N-n}{N} + \frac{2}{N}\left(\frac{n(n+1)}{2}\right) - \frac{(n-1)}{N} = \\ &= 2\frac{N-n}{N} + \frac{n(n+1)}{N} - \frac{(n-1)}{N} \end{aligned} \tag{3.24}$$

The result shows that by increasing N, next an $n = const.$, $A(N,a)$ tends to 2:

$$lim_{N\to\infty} A(N,a) = 2. \tag{3.25}$$

The proof of Theorem 3.4.4 is completed. □

Completely Balanced Arrangement

Two scenarios are considered:

- The increasing lengths of intervals are uniformly distributed across the tree,
- The distribution of lengths follows an exponential distribution.

The first case is fairly simple. Since the number of nodes is fixed with value n, it is easy to see the following.

Theorem 3.4.5. considering a completely balanced IMBT the average cost of search operation is such a constant, which is proportional with the logarithm of n. Based on Equation (3.11b), considering that $a >> 0$ (since a increasing infinitely) we get:

$$A(N,a) \approx C(n) - 1. \tag{3.26}$$

From the formula it is visible that, just like in case of Theorem 3.4.3, the $A(N,a)$ is independent from N.
The other case is a little bit trickier: it depends on the length distribution. Suppose that the nodes with smaller key values hold the longer intervals, while nodes with the highest key values hold the shorter intervals. Additionally, the rightmost interval is always one. Compensating this constraint without increasing the number of nodes the leftmost

interval always absorbs the surplus. We do not give the related formula and the associated deduction here, but consider the construction of a similar, however, "more realistic" arrangement in the next subsection.

The Theorem 3.4.5 can be easily proven based on Equation (3.11b):

$$A(N,a)\frac{3}{2}log_2\Big(\frac{N}{a}+1\Big)+\frac{3a}{2N}log_2\Big(\frac{N}{a}+1\Big)-\frac{a+1}{a} \tag{3.27}$$

By replacing $\frac{N}{a}$ with n, and considering that during the examinations $n = const.$:

$$\begin{aligned} A(n=const.,a) &= \frac{3}{2}log_2\Big(n+1\Big)+\frac{3}{2n}log_2\Big(n+1\Big)-\frac{a+1}{a}= \\ &= c_1+c_2-\frac{a+1}{a}. \end{aligned} \tag{3.28}$$

As $a >> 0$ (since a is increasing infinitely) we get the following approximation:

$$A(N,a)\approx C(n)-1.\square \tag{3.29}$$

During the proofs above, we restricted our examinations to such cases where the requested key has already stored in the data structure. Of course the missing keys would modify the results: the interval length of the gaps should be considered, instead of the interval length of the nodes. Here, the weight of the half open intervals should be handled by care.

An Exception: Completely Balanced Arrangement, Temporary Gaps, Infinite Nodes and Increasing Average

In this subsection, we introduce an arrangement, where none of the two criteria from Section 2 hold: the average length of the intervals is increasing, along with the increasing number of nodes. Our initial assumption is that the interval lengths are exponential according to power of two. Additionally the longest interval has the smallest left-hand key value, the second longest interval has the second smallest left-hand key value and so on. Moreover we suppose that the length of the shortest interval is always 2^0. Since we are examining asymptotic results we apply the simplification that only the right side distances will be taken into account for the determination of the full weight of IMBT. That is, if the length of an interval is $l_i(=2^i)$ then, with this approach we weight the right distances of a node by the full l_i, instead of $l_i-1(=2^i-1)$. However, it is easy to see that as $N\to\infty$ this difference becomes insignificant. By considering a tree nodes arrangement and applying the above constraints we obtain interval lengths of $2^0, 2^1, 2^2$. As we stipulated before, the longest interval has the smallest left key value. Therefore, in a balanced IMBT 2^2 interval has the distance from the root 3 comparisons (we take into account the right hand distances). The 2^1 interval is the root node, therefore, we count with 2 comparisons. The 1 key interval is in the right side of the balanced IMBT, therefore to reach the majority of that keys requires 4 comparisons. As we stated before, during the calculations we assumed that the following conditions are hold:

- Any key can be the subject of the search operation with equal probability,
- The key is already in the tree.

Therefore, during the determination of the average cost of search operation, the actually expected value of comparisons is calculated. According to our assumption the probability of we are looking for key k is $1/N$, where $k \in \{1...N\}$. Since during the j^{th} comparison several keys can be found the j^{th} comparisons has to be weighted with the length of the intervals. According to the above mentioned the $A(N)$ average cost is $1/N$ multiplied by the sum of weighted nodes, where a particular weight, which belongs to a single node is the multiplication of the distance and the length of the interval. The sum of weighted nodes, that is the total weight, is denoted by TW.
Let us assign the s_1, s_2 and s_3 to the above numbers, respectively. That is, $s_1 = 3, s_2 = 2$ and $s_3 = 4$. The approximate value of the total weight of the tree is the following:

$$TW = s_1 2^0 + s_2 2^1 + s_3 2^2 \tag{3.30}$$

Now by extending our examination to a $n = 7$ nodes arrangement, the longest interval is 2^6. The comparison weights depend on the lengths and the distances from the root, therefore in this case we obtain:

$$TW = (s_1+2)2^0+(s_2+2)2^1+(s_3+2)2^2+(s_1+1)2^{0+4}+(s_2+1)2^{1+4}+(s_3+1)2^{2+4}+(s_2+0)2^3 \tag{3.31}$$

From the above two equations we can formulate the recursive extension/composition rule: Take the given expression which is valid for n nodes. To determine the $2n + 1$ nodes arrangement, first copy the whole formula and increase by 2 the multipliers of the powers. Then add the formula with the multipliers increased with one and powers increased by 2. According to modification 2, increase the multiplier values by 1. Additionally, add the corresponding base 2 value to the exponents. Finally add the missing new root member to the expression. To make it more understandable, here we give an extension of Equation (3.31).

$$\begin{aligned} TW = {} & (s_1 + 2 + 2)2^0 + (s_2 + 2 + 2)2^1 + (s_3 + 2 + 2)2^2 + \\ & + (s_2 + 0 + 2)2^3 + (s_1 + 1 + 2)2^{0+4} + (s_2 + 1 + 2)2^{1+4} + (s_3 + 1 + 2)2^{2+4} + \\ & + (s_1 + 2 + 1)2^{0+8} + (s_2 + 2 + 1)2^{1+8} + (s_3 + 2 + 1)2^{2+8} + \\ & + (s_2 + 0 + 1)2^{3+8} + (s_1 + 1 + 1)2^{0+4+8} + (s_2 + 1 + 1)2^{1+4+8} + (s_3 + 1 + 1)2^{2+4+8} + \\ & + (s_2 + 0 + 0)2^7. \end{aligned} \tag{3.32}$$

This is such a recursive rule, that it affects both the multipliers and the exponents. The related contingency table is shown in Figure 3.15. In the figure, we indicated the number of steps that are required to achieve the interval opening keys, instead of the closing. That is, every weight associated values in the column are shifted by one. Based on the figure and Equation (3.32) we can say that

$$\begin{aligned} & d(w_1) = 1, \ where \ w_1 = (s_2 + 0 + 0) = 2 \\ & d(w_2) = 1, \ where \ w_2 = (s_2 + 0 + 1) = 3 \\ & d(w_3) = 2, \ where \ w_3 = (s_2 + 0 + 2) = (s_2 + 1 + 1) = 4 \\ & d(w_4) = 3, \ where \ w_4 = (s_2 + 1 + 2) = (s_2 + 2 + 1) = (s_1 + 1 + 1) = 5 \end{aligned} \tag{3.33}$$

etc.
Since the average length of the intervals is strictly tied to N, the average cost of search operation is depending exclusively on N, that is

$$A(N) = \frac{1}{N} \times TW_N. \tag{3.34}$$

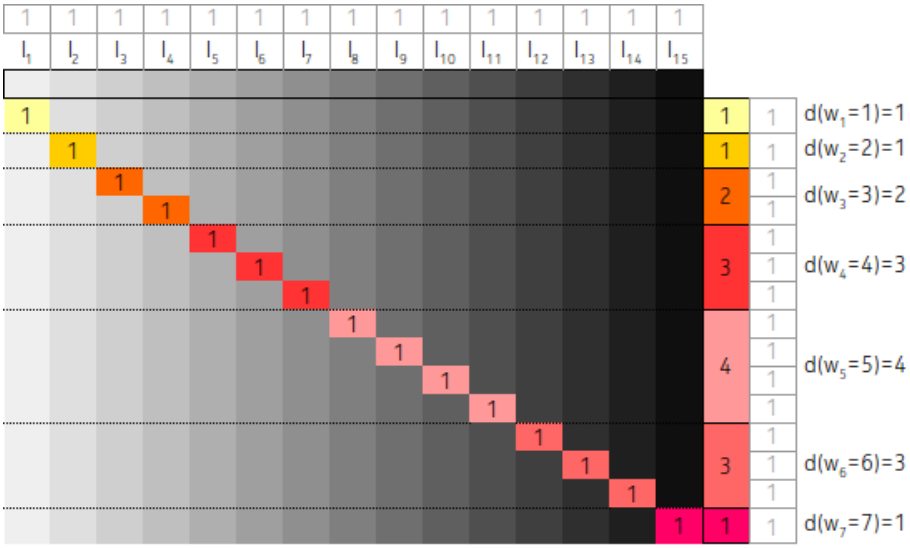

Figure 3.15: The contingency tables of IMBT where all the interval lengths are different.

The section relates to *Thesis 4.2.2*.

3.5 Arbitrary Distribution - The Matrix Representation

Even though the tree can be made fully balanced, there is no control on the sizes of the intervals which depend entirely on the arrival pattern. In other cases such formulas exist but are difficult to evaluate for instance: such is the case when the interval lengths are in a geometric progression, for which the closed formula is hard to evaluate and it does not give any insight regarding the computational complexity. In order to gain a formula

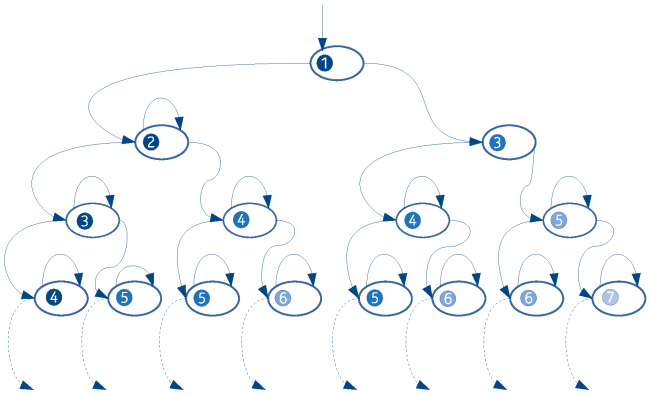

Figure 3.16: IMBT coloured distribution of traversal related weights

for a generic case, I will introduce a formula for constant interval sizes and generalize further from that. To evaluate the complexity of search for constant interval sizes, we propose to count the number of distinct comparison operations for each node in the tree according to Fig.3.16, assuming the search stops in the respective node. Then we weigh the number of comparisons for the particular nodes with the size of the interval covered

by the particular node, assuming that the likelihood for the search to end in one of the nodes is proportional with the size of the respective interval.

We can arbitrarily choose the left or the right side of the intervals. Let's first choose their left side.

In Fig. 3.17 we show the cardinality of distinct comparison operation counts per node for the levels of a fully balanced tree. It turns out that the distinct counts are exactly the

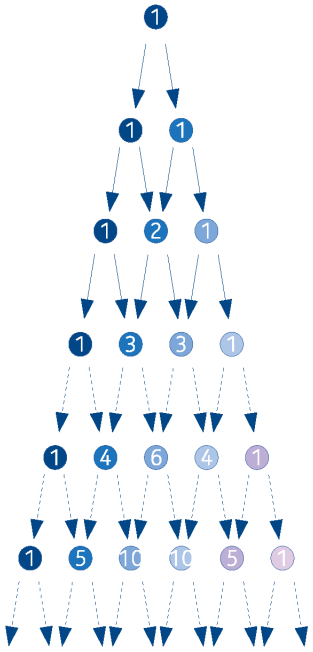

Figure 3.17: Binomial distribution of the traversal related weights in IMBT

numbers from Pascal's triangle, in the increasing order of the levels of the tree. Therefore the numbers of distinct counts are exactly the binomial coefficients:

$$\sum_{k=0}^{h}\sum_{j=0}^{k}\left(\frac{k!}{j!(k-j)!}\right)=\sum_{k=0}^{h}\sum_{j=0}^{k}\binom{k}{j}, \tag{3.35}$$

supposing that the root level is marked by zero. Assuming intervals of size one, when in fact the left and right hand comparisons fall into one single comparison, the distinct numbers of counts need to be weighed by the counts themselves, therefore the total weight becomes:

$$TW_h=\sum_{k=0}^{h}\sum_{j=0}^{k}\binom{k}{j}(j+k+1). \tag{3.36}$$

Supposing that the interval lengths can be characterized with an average value a, then we can rewrite the formula into the following one:

$$TW_h=\sum_{k=0}^{h}\sum_{j=0}^{k}\binom{k}{j}(j+k+1)(a-1). \tag{3.37}$$

As the intervals increase with the arrival of packets, the right side comparisons in the nodes will be with a factor of the interval size - 1 more times executed than not, therefore

the total weight can be better approximated with:

$$TW_h = \sum_{k=0}^{h} \sum_{j=0}^{k} \binom{k}{j} (j + k + 2)(a - 1). \tag{3.38}$$

However, in this form of the formula it is still impossible to express the search complexity for intervals with arbitrary lengths.

3.5.1 The Matrix Representation

In the following we will transform the total weight to a format suitable for expressing it in a matrix form that will allow us to consider intervals of arbitrary length. We consider again the left sides of the intervals. The numbers of distinct counts in the first level can be rewritten as:

$$\sum_{i=2}^{3} i = \mathbf{1} \times 2 + \mathbf{1} \times 3 = 5 \tag{3.39}$$

The second level can be rewritten as:

$$\sum_{j=1}^{2} \sum_{i=2}^{3} (j + i) = \mathbf{1} \times 3 + \mathbf{2} \times 4 + \mathbf{1} \times 5 \tag{3.40}$$

The corresponding expressions for the 3rd and 4th level are:

$$\sum_{k=1}^{2} \sum_{j=1}^{2} \sum_{i=2}^{3} (k + j + i) = \mathbf{1} \times 4 + \mathbf{3} \times 5 + \mathbf{3} \times 6 + \mathbf{1} \times 7 \tag{3.41}$$

$$\sum_{l=1}^{2} \sum_{k=1}^{2} \sum_{j=1}^{2} \sum_{i=2}^{3} (l + k + j + i) = \mathbf{1} \times 5 + \mathbf{4} \times 6 + \mathbf{6} \times 7 + \mathbf{4} \times 8 + \mathbf{1} \times 9 \tag{3.42}$$

From the pattern we can recognize that this representation still follows the binomial coefficients regarding the distributions of the weights. Let's apply the following equivalent transformations:

$$\sum_{i=2}^{3} i = \mathbf{1} \times 2^{1} + \mathbf{1} \times 3^{1} = 5 \tag{3.43}$$

$$\sum_{j=1}^{2} \sum_{i=2}^{3} (j + i) = \mathbf{5} \times 2^{1} + \mathbf{3} \times 2^{1} \tag{3.44}$$

$$\sum_{k=1}^{2} \sum_{j=1}^{2} \sum_{i=2}^{3} (k + j + i) = \mathbf{5} \times 2^{2} + \mathbf{3} \times 2^{2} + \mathbf{3} \times 2^{2} \tag{3.45}$$

$$\sum_{l=1}^{2} \sum_{k=1}^{2} \sum_{j=1}^{2} \sum_{i=2}^{3} (l + k + j + i) = \mathbf{5} \times 2^{3} + \mathbf{3} \times 2^{3} + \mathbf{3} \times 2^{3} + \mathbf{3} \times 2^{3} \tag{3.46}$$

We can recognize the following generalized rule from the transformations above:

$$W_j = (5 + (j-1)3)2^{j-1}, \tag{3.47}$$

where W_j is the weight related to level j of the tree and root is level 0.

$$TW_h = 1 + \sum_{i=2}^{h} (5 + (i-2)3)2^{(i-2)}. \tag{3.48}$$

And finally, considering interval lengths bigger than one:

$$TW_h = (a-1)\left(1 + \sum_{i=2}^{h} (5 + (i-2)3)2^{(i-2)}\right) \tag{3.49}$$

(3.43), (3.44), (3.45) can be rewritten in a matrix form, as described below:

$$W_1 = \begin{bmatrix} 1 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 1 \end{bmatrix} \tag{3.50}$$

$$W_2 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 2 \\ 2 & 3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{3.51}$$

$$W_3 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \\ 2 & 1 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 2 \\ 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \tag{3.52}$$

We introduce the following notations: the first factor is called *interval length vector* and the second factor is called *weight matrix*. Additionally let $\mathbf{1}_p = [1_1\ 1_2\ ...\ 1_p]^T$. In this new representation we can recognize that the weight matrix corresponding to level k of the IMBT can be evaluated with the following recursive formula:

$$\mathbf{W}_{k+1} = \begin{bmatrix} \mathbf{1}_{2^k} & \mathbf{W}_k \\ 2\mathbf{1}_{2^k} & \mathbf{W}_k \end{bmatrix} \tag{3.53}$$

To generalize from intervals of length 1 to intervals of length a across all nodes, the in interval length can be introduced in the interval length vector (ILV) in the following way, for the 2nd level of the tree taken as an example:

$$\begin{bmatrix} (a-1) & (a-1) & (a-1) & (a-1) \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 2 \\ 2 & 3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{3.54}$$

At this point we can generalize from intervals of constant length to arbitrary intervals at every level and in every individual node by replacing the element of the interval length

vector at the particular level with the appropriate interval length. As an example we express the geometric progression based interval lengths with the help of the matrix representation.

$$
\begin{aligned}
&TW_3 = \\
&= \begin{bmatrix} 2^{14} & 2^{12} & 2^{10} & 2^8 & 2^6 & 2^4 & 2^2 & 2^0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \\ 2 & 1 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 2 \\ 2 & 2 & 3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} + \\
&+ \begin{bmatrix} 2^{13} & 2^9 & 2^5 & 2^1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 2 \\ 2 & 3 \end{bmatrix} \times \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \\
&+ \begin{bmatrix} 2^{11} & 2^3 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} \times \begin{bmatrix} 1 \end{bmatrix} + \\
&+ 2^7.
\end{aligned} \tag{3.55}
$$

The Total Weight can be expressed in a more compact form if we use *interval matrices* (IM) instead of interval length vectors, along with utilizing only the weight matrix of order k. Interval matrices can be defined as the Hadamard product (or element-wise product '$\circ$') between a coefficient matrix and the interval vectors arranged in a convenient form, the *interval arranged matrix* (*IVA*). The *coefficient matrix* has with one less as many rows as there are levels in the IMBT. The first row consists of ones. The first half of the second row consists of 2-s, the second half consists of -1-s. The third row has 2-s and -1-s in the first and fourth quarter, the rest is 0-s. The next row has 2-s and -1-s in the first and last eighth and zeros in the rest, and so on. Finally the last row consists of two 2-s at the beginning, two -1-s at the end and zeros in the middle. The *coefficient matrix* is a $k-1$ by 2^{k-1} matrix, where k is: (the number of levels in the IMBT) - 1.

$$
C_k = \begin{bmatrix} & & \mathbf{1}_{2^{k-1}}{}^T & \\ 2 \times \mathbf{1}_{2^{k-2}}{}^T & & & -1 \times \mathbf{1}_{2^{k-2}}{}^T \\ 2 \times \mathbf{1}_{2^{k-3}}{}^T & \mathbf{0}_{2^{k-2}}{}^T & & -1 \times \mathbf{1}_{2^{k-3}}{}^T \\ & & \cdots & \\ 2 & 2 & \mathbf{0}_{2^{k-1}-4}{}^T & -1 & -1 \end{bmatrix} \tag{3.56}
$$

The *IVA* matrix is a $k-1$ by 2^{k-1} matrix like the coefficient matrix, in which the interval length vectors are represented in the following way: the interval length vector of the last level represents the first row, the second row consists of the interval length vector of the previous level written twice and so on. The last row contains the first interval length vector, representing the level below the root, written $(k-1)/2$ times, as below:

$$
IVA_k = \begin{bmatrix} & & ILV_{k-1} & \\ & ILV_{k-2} & ILV_{k-2} & \\ . & . & . & \\ ILV_1 & ILV_1 & ... & ILV_1 \end{bmatrix} \tag{3.57}
$$

The *characteristic matrix* is obtained as the Hadamard product of the coefficient matrix and the interval arranged matrix, multiplied with the weight matrix with an ordinary matrix multiplication:

$$Ch_k = (C_k \circ IVA_k) \times W_k = IM_k \times W_k \tag{3.58}$$

The characteristic matrices of balanced IMBT-s with interval length 1 with 2, 3 and 4 levels are shown in the following:

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 3 \end{bmatrix} = \begin{bmatrix} 5 \end{bmatrix} \tag{3.59}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 2 & -1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 1 & 3 \\ 2 & 2 \\ 2 & 3 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 0 & 5 \end{bmatrix} \tag{3.60}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & -1 & -1 & -1 & -1 \\ 2 & 2 & 0 & 0 & 0 & 0 & -1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 & 2 \\ 1 & 1 & 3 \\ 1 & 2 & 2 \\ 1 & 2 & 3 \\ 2 & 1 & 2 \\ 2 & 1 & 3 \\ 2 & 2 & 2 \\ 2 & 2 & 3 \end{bmatrix} = \begin{bmatrix} 12 & 12 & 20 \\ 0 & 6 & 10 \\ 0 & 0 & 5 \end{bmatrix} \tag{3.61}$$

For IMBT-s discussed earlier with interval lengths showing an exponential distribution the respective coefficient matrices are:

$$\begin{bmatrix} 2^2 & 2^0 \end{bmatrix}$$

$$\begin{bmatrix} 2^6 & 2^4 & 2^2 & 2^0 \\ 2 \times 2^5 & 2 \times 2^1 & -2^5 & -2^1 \end{bmatrix} \tag{3.62}$$

$$\begin{bmatrix} 2^{14} & 2^{12} & 2^{10} & 2^8 & 2^6 & 2^4 & 2^2 & 2^0 \\ 2 \times 2^{13} & 2 \times 2^9 & 2 \times 2^5 & 2 \times 2^1 & -2^{13} & -2^9 & -2^5 & -2^1 \\ 2 \times 2^{11} & 2 \times 2^3 & 0 & 0 & 0 & 0 & -2^{11} & -2^3 \end{bmatrix}$$

and the (3.62) related characteristic matrices up to size 5 are shown below:

$$
\begin{bmatrix} 11 \end{bmatrix}
$$

$$
\begin{bmatrix} 90 & 187 \\ 0 & 70 \end{bmatrix}
$$

$$
\begin{bmatrix} 21930 & 23130 & 48059 \\ 0 & 8772 & 17990 \\ 0 & 0 & 4120 \end{bmatrix} \tag{3.63}
$$

$$
\begin{bmatrix} 1431677610 & 1437226410 & 1515870810 & 3149642683 \\ 0 & 572671044 & 574890564 & 1179010630 \\ 0 & 0 & 134746128 & 270012440 \\ 0 & 0 & 0 & 16777600 \end{bmatrix}
$$

The characteristic matrix, despite being only a snapshot of the state of the tree at a particular time instant, represents a standalone piece of information that can be utilized to analyze the behavior of the tree.

3.5.2 Model Refinements

As the intervals increase with the arrival of packets, as mentioned earlier, the right side comparisons in the nodes will be almost always executed - in fact with a factor of the interval size - 1 more times executed than not. To express this in the model we have to replace the initial *'2'* to *'3'* and the *'3'* to *'4'* in (3.53) and thus we arrive to the following:

$$
\begin{bmatrix} 3 \\ 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 3 \\ 1 & 4 \\ 2 & 3 \\ 2 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & 3 \\ 1 & 1 & 4 \\ 1 & 2 & 3 \\ 1 & 2 & 4 \\ 2 & 1 & 3 \\ 2 & 1 & 4 \\ 2 & 2 & 3 \\ 2 & 2 & 4 \end{bmatrix} \tag{3.64}
$$

The traversal strategy of the tree is the order in which we go from node to node and the order in which we check against the left side and the right side of the intervals the concrete key we are looking for. In this context we differentiate left-side-first strategies, when the left side of the intervals is checked against first, and right-side-first strategies, when the right hand of the intervals is checked against first. We additionally assume that the tree is traversed from up towards the bottom and from left to right, as shown in Fig.3.16. Traversal strategies may be important means to increase the efficiency of accessing the tree, especially in the case under discussion, in which packets usually arrive in the increasing order of their sequence number, except for some outliers. In such a case, when we additionally assume all the keys bigger than a defined smallest sequence number, the right-side-first traversal is more efficient. The right-side-first traversal strategy can be

encoded in the weight matrices under the form of a horizontal mirroring:

$$\begin{bmatrix}4\\3\end{bmatrix} \Rightarrow \begin{bmatrix}2 & 4\\2 & 3\\1 & 4\\1 & 3\end{bmatrix} \Rightarrow \begin{bmatrix}2 & 2 & 4\\2 & 2 & 3\\2 & 1 & 4\\2 & 1 & 3\\1 & 2 & 4\\1 & 2 & 3\\1 & 1 & 4\\1 & 1 & 3\end{bmatrix} \tag{3.65}$$

In this case for the correct computations, the interval coefficient matrices need to be vertically mirrored. Another effect of mostly in-order arrival is that the tree will not be perfectly balanced. This effect be taken into account through virtually balancing the tree with nodes having interval length of zero. In the matrix representation this can be translated into height supplemented matrices. This makes the analysis more convenient compared to previous approaches. An example of virtually balanced tree with virtual intervals of length 0 is shown in Fig.3.18.a). In the figure we show only the lengths of the intervals $L_1, L_2, L_3, ...$ rather than their left and right extremes. The supplemented

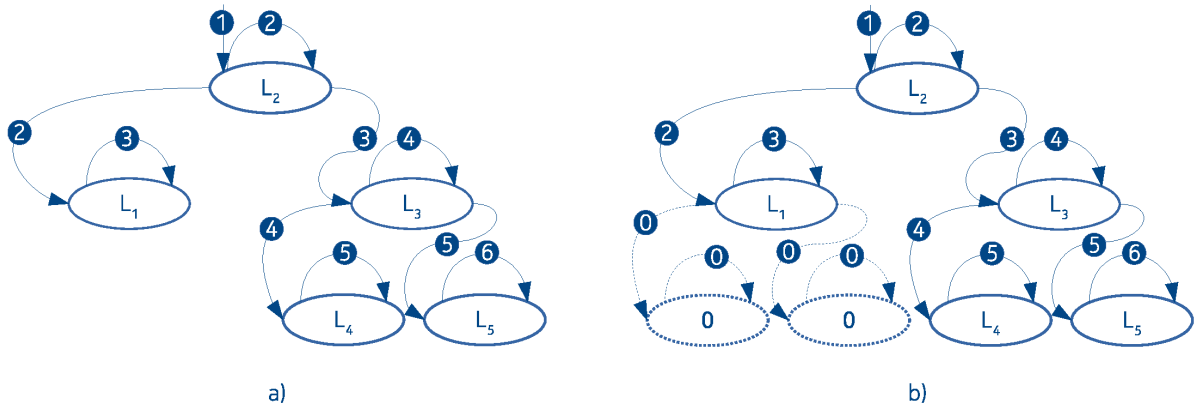

Figure 3.18: a) IMBT balancing imperfection in incremental environment. b) Supplemented IMBT for equivalent numerical simulations

IMBT is shown in Fig.3.18.b). In the matrix description the virtual intervals of length 0 are encoded into the interval length matrix.

$$\begin{bmatrix}0 & 0 & 1 & 1\\2 & 2 & -1 & -1\end{bmatrix} \times \begin{bmatrix}1 & 3\\1 & 4\\2 & 3\\2 & 4\end{bmatrix} \tag{3.66}$$

$$\begin{bmatrix}0 & 0 & 0 & 0 & 1 & 1 & 1 & 1\\2 & 2 & 2 & 2 & -1 & -1 & -1 & -1\\2 & 2 & 0 & 0 & 0 & 0 & -1 & -1\end{bmatrix} \times \begin{bmatrix}1 & 1 & 3\\1 & 1 & 4\\1 & 2 & 3\\1 & 2 & 4\\2 & 1 & 3\\2 & 1 & 4\\2 & 2 & 3\\2 & 2 & 4\end{bmatrix} \tag{3.67}$$

An additional advantage of the matrix representation relates to the mirroring of the interval lengths in the tree. If we assume the distribution of the interval lengths to still following the geometric progression, but for some reason the keys will arrive in decreasing order, then the nodes on the left side of the tree will cover shorter intervals than the ones on the right side. This change can be easily expressed by swapping the order in which the coefficients are inserted into the interval length tree, as shown below:

$$\begin{bmatrix} 2^0 & 2^2 \end{bmatrix}$$

$$\begin{bmatrix} 2^0 & 2^2 & 2^4 & 2^6 \\ 2 \times 2^1 & 2 \times 2^5 & -2^1 & -2^5 \end{bmatrix} \tag{3.68}$$

$$\begin{bmatrix} 2^0 & 2^2 & 2^4 & 2^6 & 2^8 & 2^{10} & 2^{12} & 2^{14} \\ 2 \times 2^1 & 2 \times 2^5 & 2 \times 2^9 & 2 \times 2^{13} & -2^1 & -2^5 & -2^9 & -2^{13} \\ 2 \times 2^3 & 2 \times 2^{11} & 0 & 0 & 0 & 0 & -2^3 & -2^{11} \end{bmatrix}$$

3.5.3 Experimentation results

In the following we compare the results of the closed formula (3.11b) with the results from numerical simulations for the total number of keys taken from the following set: $N = 2^n - 1$, where n represents the number of nodes in the balanced IMBT taken from the set $n = 3, 7, 15, 31$ respectively. In case of the closed formula the average interval lengths are $a = 2.33$, 18.14, 2184.46, 69273666.03. We considered both traversal variants, so (3.53) was the starting point, where the weight matrices from both (3.52) and (3.64), were considered. As a starting point we considered intervals of constant length. In the next step we evaluated the search complexities for interval lengths that represent a geometric series. Matrices of the form (3.68) are referred to as right weighted and those from (3.62) are referred as left weighted. The evaluated search complexities according to the distinct approximations described earlier are summarized in Table - 3.3, where 'lsh' refers to left-side-heavy, 'rsh' refers to right-side-heavy and 'ave' refers to uniform interval length distributions. 2-3 and 3-4 denote left- and right-hand-side first traversals. It is visible that

Table 3.3: Comparison of Formula (3.11b) and Matrix based computations

	N = 7	**N = 127**	**N = 32767**	**N = 2147483647**
Nodes	3	7	15	31
Approximate formula	2.37	4.21	5.86	7.43
2-3 lsh	1.85	2.79	3.78	4.78
2-3 ave	1.99	3.14	4.39	5.74
2-3 rsh	2.28	4.02	6.00	7.99
3-4 lsh	2.57	3.79	4.78	5.78
3-4 ave	2.99	4.14	5.39	6.74
3-4 rsh	3.28	5.02	7.00	8.99

the search complexity has a logarithmic dependence on the number of nodes. It can also

be seen that the traversal strategy has a significant effect on the search complexity, up to almost 50%. In the following we evaluate the search complexity in the case when interval lengths follow the geometric progression. This might occur in realistic scenarios, where for instance there is a temporary packet loss in the network. In order to formulate more general conclusions we have chosen several different bases for the geometric progressions. The series of intervals were selected/generated according to the following procedure: we chose the following set of bases, $B = \{b_i\} = \{1.2, 1.5, 2, 2.2, 2.5, 3, 3.2, 3.5, 4, 4.2, 4.5, 5, 5.2, 5.5, 6\}$ and the set of possible number of nodes, $n = \{3, 7, 15, 31, 63, 127, 255\}$. We chose the total number of keys from a set N_i of size 1000 but spanning several orders of magnitude, $N_i \in \{1.0912^1, 1.0912^2, \ldots 1.0912^{1000}\}$. The procedure of selecting the concrete values for N_i were such that the following condition is satisfied with respect to 2 consecutive values from the set B: $\frac{N_i \times (b_i - 1)}{b_i^{n_i}} \geq 1$ and $\frac{N_i \times (b_i - 1)}{b_i^{n_i+1}} < 1$. By choosing this selection the asymptotic deviation from the original N tends to zero but still the IMBTs can be considered balanced. The numbers from the set B are not integers but in practical cases interval sizes are multiples of tens of the base vectors, therefore this does not matter. In Fig.3.19 the average cost of search operation is presented for interval

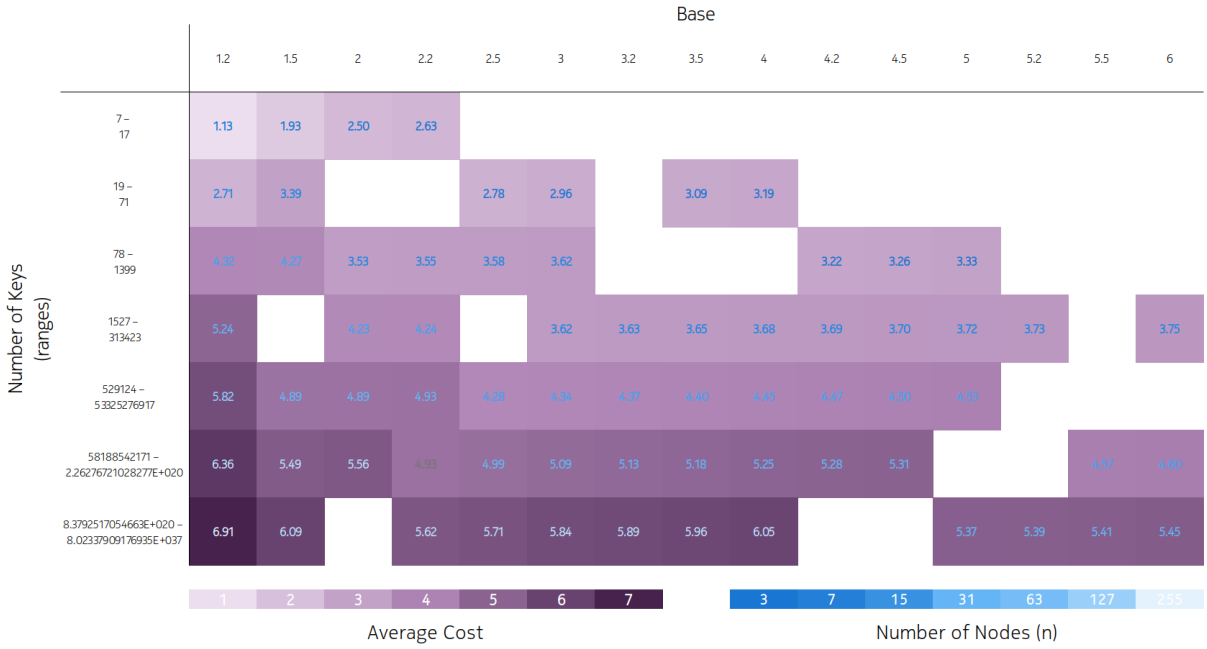

Figure 3.19: Node cardinality and the cost of search as a function of the base of the geometric progression. Darker areas indicate higher search operation cost. The lighter numbers indicate more nodes

sizes in a geometric progression as a function of the base of the geometric progression and the number of nodes. Darker areas indicate more elevated search costs. These typically occur for cases when intervals are shorter and more nodes are required to store the same number of keys. The white/empty areas are gaps for which the procedure did not find interval series that match the selection criteria.
In Fig.3.20 we present more detailed results for the lower and upper bound of the base values and a value from the middle. The figures show the number of nodes as a function of the total number of keys. The diameter of the circles is proportional with the cost of search for the particular balanced IMBT instance. From figures 3.20 a) - c) it is visible, that for interval sizes that represent geometric progressions the relation between N and

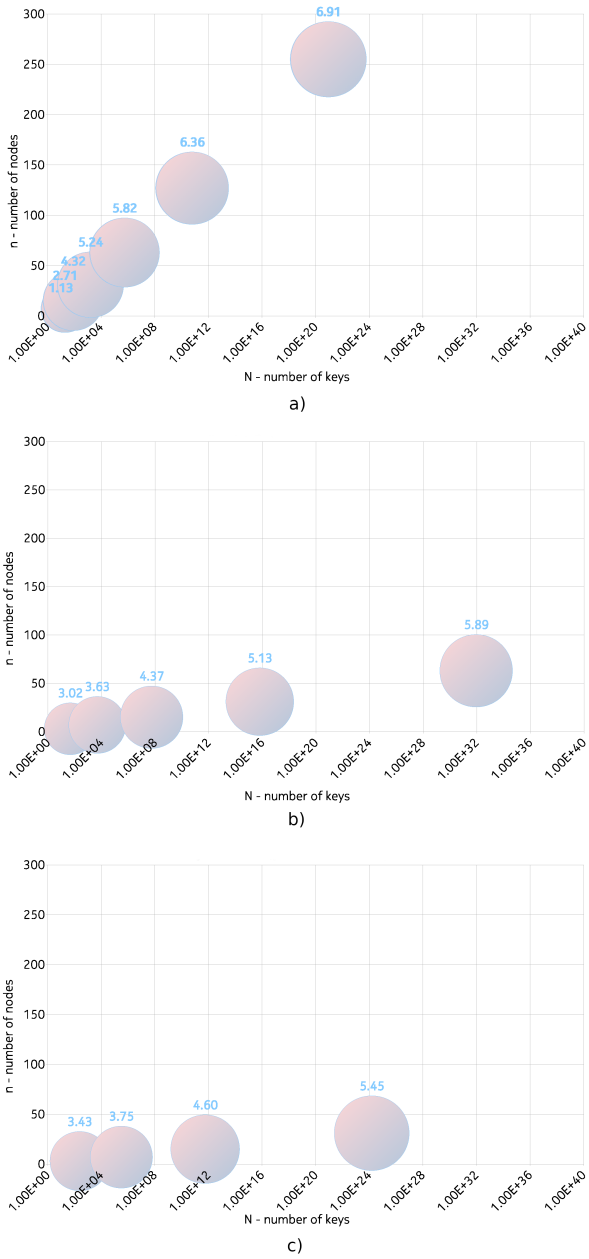

Figure 3.20: Number of nodes and cost of search for geometric progression with a) base = 1.2 b) base = 3.2 and c) base = 6

n can be expressed by $n(N) = log_b(N)$. Additionally the relation between the number of nodes n and the cost of search operation A is $A(n) = log_2(n)$. That is, the trend is:

$$A(N) = log_2(log_b(N)) \tag{3.69}$$

It can be seen that the slope of the dependency decreases with the increasing b base of the geometric progression.
We can formulate now the following theorem.

Theorem 3.5.1. *consider a snapshot about the left weighted geometric progression based interval lengths with instantaneous 'b' base. Then there is always a series of logical time dependent $b(t)$, where $b(t_0) = b$, such that the instantaneous 'n' number of nodes become constant, therefore 'A(N)' is upper bounded.*

Proof. To show that, we replace 't' with 'N', as a 'logical' time, and rewrite the logarithm in the following equivalent form:

$$\begin{aligned} log_b(N) &= \frac{log_x(N)}{log_x(b)} = n = const. \\ log_x(b) &= \frac{log_x(N)}{const.} = \frac{log_x(N)}{n}, \\ b_e(N) &= x^{\frac{log_x(N)}{const.}} = x^{\frac{log_x(N)}{n}}, \end{aligned} \tag{3.70}$$

where x is arbitrary. That is $A(N) \leq 2 \times log_2(log_{b_e(N)}(N)) + 1 = 2 \times log_2(n = const.) + 1 = c_1$. □ □

Consequence: Let $b(N)$ the values of bases over time of an IMBT. Based on the $b_e(N)$ value the following cases should be examined:

1. if the $d(N) = |b_e(N) - b(N)| \approx 0$, that is, the distance stochastically converges to zero,
2. if the $d(N) = |b_e(N) - b(N)| \approx |\sqrt[n_e]{N} - \sqrt[n_{ne}]{N}|$,
3. if is there always an N_i such that $d(N)$ tends to ∞ faster than the $|\sqrt[n_e]{N} - \sqrt[n_{ne}]{N}|$ function for any two arbitrary selected but fixed n_e, n_{ne}.

The first case represents a **dynamic equilibrium**: the number of nodes fluctuates around an $n = const.$ During the second case another, n_{ne} dynamic equilibrium exists, which differs from the instantaneous n_e. According to the third case there is no dynamic equilibrium: there is always a threshold N_i such that $N >> N_i$ then $d(N)$ tends to ∞ faster than the $|\sqrt[n_e]{N} - \sqrt[n_{ne}]{N}|$ for any two arbitrary but fixed n_e, n_{ne}.
We can generalize Theorem-3.5.1, and give a sufficient condition regarding the dynamic behavior of the tree. Let's consider all the nodes in an IMBT with arbitrary interval length distribution. Sort them ascending/descending order regarding their interval length independently from their original position in the tree.

Theorem 3.5.2. *If the ratio of the interval length ordered nodes can be statistically proximated with the series of N dependent $b(N)$ based geometric progressions such that $d(N) = |b_e(N) - b(N)| \approx 0$, that is $d(N)$ stochastically converges to zero, then the tree is in a dynamic equilibrium state.*

The proof is identical with the previous one.

The section relates to *Thesis 4.2.4*.

3.6 Packet De-duplication in Distributed Environment

Stream processors, like Storm [6] or Flink [65], are frequently employed in distributed, computing-intense applications with data parallelism and data locality features. They are

typically high throughput, long-running systems, therefore they need to automatically or semi-automatically scale and handle processing node failures.

In horizontal scale-out new nodes are added to the distributed system and the system state is synchronized from incumbent nodes to the nodes added in the scale-out procedure [66]. Failure scenarios are mitigated via replicas that help additionally with the distribution of the load between instances holding the same replica [67]. Between replicas a synchronization is needed. So synchronization is required both for scale-out and failure mitigation procedures.

We assume a long running application in which streams of *(key, value)* pairs are processed by a distributed system, typically a stream processing engine, and stored in a persistent storage system and there needs to be a guarantee that a particular key-value tuple is not stored twice.

Duplication can be avoided at the time of insertion to the storage. A well known and widely applied data structure, which makes possible the filtering during the insertion is called *Merkle-Tree* [68], which is a hash of hashes and the basis of many distributed key-value store. Another insertion time duplication filter method and data structure is the *Log Structured Merge - LSM*, [69], where a so called out-of-place update procedure is implemented. This solution is very popular in modern data stores, like Big Table, HBase, Apache Cassandra, etc. Since the original implementation several more optimized variants are appeared, like bLSM [70], where the authors through a combination with a *B-tree* are enhanced the speed of read operation.

Insertion-time avoidance can be expensive, therefore duplication avoidance can take place in a logical layer preceding the insertion. There are various possible solutions for pre-insertion duplication avoidance. One solution could be storing the keys upon arrival in an in-memory Distributed Hash Table (DHT). One realization is the so-called CHORD [61]. This solution works up to a point where the increasing storage requirement and the possible need to re-hash become a bottleneck. In my initially examined case the keys were composed of a fixed number of attributes (attribute-based naming), where we could set up a distributed, rule based association method such that the same sequence number being allocated to the same key, independently on the concrete node performing the association. This is a simpler sort of linearisation compared to e.g. the Hilbert space-filling curve [71]

Here I extend the concept of IMBT to a distributed scenario, where eventual consistency [72] is provided.

Regarding synchronization from the naive *message flooding* approach (every incoming key triggers a broadcast message) several sophisticated methods have been worked out for (key) synchronisation purposes, [73], [74]. However, the conditions are required for optimal operation in the indicated references differ from our dense key-space.

3.6.1 Synchronization Methods

In the context of an Extract Transform Load (ETL) environment that usually involves one type of stream processing framework, we assume a layer of P equivalent data processing nodes with the task to persist data to a permanent storage. Each processing node receives packets with a sequence number. The previous layer is the source of data packets and chooses the data processing node based on a shuffle grouping policy, playing also the

role of a load-balancer. In this way each entity from the considered layer has a similar processing load, even if the time required to process a packet can vary between packets. Packets may arrive out of order, may get duplicated or lost in the process of transmission. In case of a grouping strategy different from shuffle grouping, the individual nodes could implement an IMBT, independently from the rest of processing nodes and the working assumption would be that sequence numbers of the packets arriving to a particular node increase at a rate of P. In case of shuffle grouping a duplicated packet may arrive to a different node than the previous instance of the same packet. Therefore an individual IMBT is not enough to filter out duplicates.

Centralized IMBT

The baseline solution is another type of processing node that keeps track of the IMBT of the entire system. We may call this node the IMBT proxy. Each processing node has to contact the IMBT proxy at every received packet to check whether the packet has arrived already or not in order to update the tree. In case when rate P is high, the IMBT proxy quickly becomes a bottleneck.

Synchronization with full IMBT

An efficient distributed solution is to circulate the full IMBT in a circular fashion so that the IMBT arrives back to the sender node after exactly P steps, that is P tree merging, Fig 3.21. To be able to *detect* the fact of duplication we have to introduce a new term,

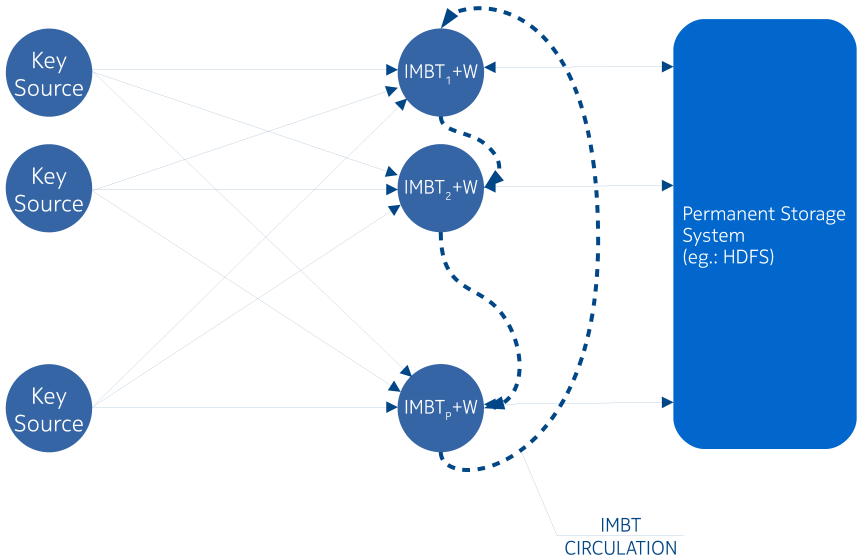

Figure 3.21: Circulating the *sync IMBT* for Synchronization Purposes

the *absorption counter*, and distinguish two types of IMBT, a node *local IMBT* and the travelling one, the so called *sync IMBT*. Every time when the *sync IMBT* arrives to a particular node two identical IMBTs will be composed (a new node *local* and a *sync IMBT*) by the merging of the two IMBTs into one. When the new *sync tree* is departing from the actual processing node the total number of keys covered by the IMBT is saved into the *absorption counter*. At the point when the IMBT instance returns to any sender

which was visited before, the received *sync tree* is merged with the *local tree* available at that point in time. If there was no duplication, the amount of overlapping keys is equivalent with the value that was stored in the *absorption counter* when the *sync tree* was sent out. If the amount of overlapping keys are higher than the *absorption counter*, it means there are packets that arrived to this node, but prior to that they arrived already to another node.

Definition 3.6.1. The period of time required for the *sync tree* to be processed exactly once by all processing nodes is called *transactional delay*.

Corollary 3.6.0.1. The *transactional delay* is the cost of a traversal between the processing nodes through a directed Hamiltonian cycle.

To be able to determine the gain provided by the IMBT approach we have to consider the followings. Let the number of keys covered by the *local* $IMBT_1$ be N_{I_1} and the number keys covered by *sync IMBT* be N_S. Let's assume that all the keys would be individually stored in a BST. According to the assumption all the keys are traveling to such nodes where all the keys are stored, that is $N_{I_1} \approx N_S$. With these condition the merging of the two BSTs would require $O(N_{I_1} + N_S)$ operations, [75]. Let's denote by v the number of vertices in an IMBT and by a the average number of *keys* covered by a vertex. Then $N = v \times a$.

Theorem 3.6.1. With the above notations the required number of steps to merge two IMBTs is $O(v_{I_1} + v_S)$.

By using the same a for both the *sync tree* and *local tree* as a rough approximation we get that $N_{I_1} = v_{I_1} \times a$ and $N_S = v_S \times a$. Therefore we can formulate the following theorem.

Theorem 3.6.2. By using IMBTs during the merging the procedure takes $a = N_{I_1}/v_{I_1}$ times less steps than by using any balanced BST, which explicitly contains all the keys.

Based on the evolution of a over time, that is $a(t)$ or $a(N)$, we can get various results regarding the gain. Since key sources are usually not perfectly ideal some of the keys are disappearing permanently and the gap remains forever in both *sync IMBT* and the *local IMBTs* as well. This fact might result a continuously increasing *sync IMBT*, which is not feasible.

Since we do not keep the whole track of the evolution of the IMBT during the transactional delay

Theorem 3.6.3. By the circulation of the full IMBT for synchronization purposes only the fact of duplication can be detected without the exact *identification* of the duplicated keys.

Due to the lack of precise *identification* we are not able to perform any kind of late correction. That is, it makes no sense to delay the write out procedure through the maintenance of a temporary buffer area in order to perform a late correction. Therefore in *Synchronization with Differential IMBT* we introduce another approach, where the time complexity might be higher, along with the same key distribution, however the duplicated keys can be exactly identified, and the size of *sync IMBT* would not increase over every limit due to permanent gaps.

Synchronization with Differential IMBT

The basic idea is to avoid the circulation of the full IMBT and circulate only the difference of the *local tree* and the incoming *sync tree* instead. The motivation behind this approach becomes evident when the gaps on the body of originally contiguous series of keys get permanent: in such cases an increasing number of nodes should circulate, in contrast with the perfectly idealistic scenario in which the number of nodes in *sync IMBT* (v_{sync}) is tending towards an e expected result, and e might be one as a best case. However, with this technique only the same (along a full Hamiltonian circle) packet duplications of 1, 2 or a multiple of 2, can be *detected* at all.In order to *detect* and *identify* every packet duplication, we propose three IMBT variants: an *active tree*, an *archive tree* and a *sync tree*. Actually the *absorption counter* is replaced by a more complex data structure, that is with an IMBT. Regarding the implementation there are at least two solutions. In the first solution both the *active* and the *archive trees* are full ones. The *archive tree* is a snapshot of the *active tree* created at the time instant when the *sync tree* leaves this node towards the downstream node. The *sync tree* sent out by this particular node towards the downstream node is the union of the difference of the *sync tree* and the *archive tree* and the difference of the *active tree* and the *archive tree* of this particular node. That is,

$$\begin{aligned} sync\_tree_{out} = (sync\_tree_{in} \setminus archive\_tree) \cup \\ (active\_tree \setminus archive\_tree) \end{aligned} \tag{3.71}$$

Assuming that keys are duplication-free, then during the union operation the subtracted trees are in principle void of overlaps, at the maximum they can have keys that are consecutive. The overlaps resulting from the union operation will indicate precisely the keys that are duplicate.

In the second implementation the *active tree* contains only the keys collected during the *transactional delay*. In this case the new *sync tree* can be evaluated subtracting the *archive tree* from the *sync tree*, merged with the result of the subtraction of *archive tree* from the *active tree*. That is,

$$\begin{aligned} sync\_tree_{out} = (sync\_tree_{in} \setminus archive\_tree) \cup \\ (active\_tree \setminus archive\_tree) \end{aligned} \tag{3.72}$$

One can realize that the same operations are performed again as in case of Formula 3.71. The new *archive tree* is equal with the merging of the *sync tree*$_{out}$ and the original *archive tree*. In parallel the *active tree* can be flushed out. The *sync tree* is sent out to the next node and the cycle of operations is repeated there. Again, just like in the previous case, the duplicated keys can be identified during the second and third operations of Formula 3.72. Therefore, in contrast to full IMBT circulation, in the implementations introduced above it might worth to maintain a temporary buffer area in order to perform late corrections preceding the write out function.

The costs of the above operations can be determined according to the followings. The role of the *sync tree* in both cases is the same. Originally it collects the newly arrived keys and spreads them to other processing nodes. If we suppose that the keys are uniformly distributed due to the load balancing, then from IMBT performance point of view the worst case would be that the keys are not consecutive at all. This situation really

may occur at the beginning, however as the *sync tree* continues visiting more and more processing nodes, the intervals are getting longer and longer, supposing that the source is really of incremental type. Then, independently from the approaches above we may use a key distribution and number of nodes dependent average length, denoted by a_{sync}. According to first implementation $v_{arch} \approx v_{act}$ and either $v_{sync} \ll v_{arch}$ or $v_{sync} \gg v_{arch}$. According to second implementation $v_{sync} \approx v_{act}$ and either $v_{sync} \ll v_{arch}$ or $v_{sync} \gg v_{arch}$. In an asymmetric environment like described above it is worth to apply the traversal based linearisation and any set operation between the data structures with approximately the same number of elements. However, between those data structures in between there is at least one magnitude of order regarding the number of elements, making IMBT-based searches much more efficient to apply. Considering the first implementation approach one can say that during the determination of $synctree_{out}$ two parallel operations can be performed. The first is the set-minus where, according to our assumption, $v_{sync} \ll v_{arch}$. Therefore the costs of the parallel operations are respectively $O(v_{sync} \times log(v_{arch}))$ and $O(v_{arch} + v_{act})$, [75]. Additionally to get $sync\_tree_{out}$ the union of the two is required, which is $O(v_{sync} + v_{sync})$. Here we used the fact that the difference between *arch tree* and *act tree* is a kind of *sync tree*. In order to ensure consistency different type of constraints appear regarding synchronization, which are out of scope of this paper.

3.6.2 Scaling

To serve as a long running process in a dynamically changing environment in terms of input load, the synchronization method should be augmented to cover scale in and scale out scenarios. In this section we will describe this.

Space Scaling

Regarding missing keys we can have two assumptions:

- The event is temporary and as time goes by the key will certainly arrive.
- The event is permanent, the key will never arrive.

If all the keys fall into the first class then, as we shown in [IF-11], it results in a binary tree with fixed number of nodes on average and a variable component, depending on the shuffle operation, with an upper bounded size. Therefore we will never not run out of space. Hence, with a fixed input load the initial setup will be able to serve the queries. In the second case, after a time depending on the average distance between the permanently missing keys, the computing node will run out of space. And, due to the total synchronization, this event will take place in all IMBT at the same time. This previously mentioned space is not necessarily determined by the size of the RAM: it can be a logical limit as well. This limit can be translated into number of vertices in the IMBT; let's denote this threshold by v_t. In the following we distinguish two type of IMBTs.

Definition 3.6.2. The IMBT, which has not exceeded its v_t limit yet, is called *mutable processing node* or *mutable IMBT*.

Definition 3.6.3. The IMBT, which has already exceeded its v_t limit, is called *immutable processing node*.

We can estimate, considering the input load and the transactional delay the number of IMBTs that can serve the filtering related parallelly executed search operations. Let us set up a C_1^* cluster with that number of mutable IMBTs and denote with k the cardinality of C_1^*. When the number of IMBT vertices exceeds v_t, a message is initiated that informs all the other IMBTs in this cluster, formed by the mutable IMBTs, that the capacity limit has been reached and, except this last synchronization message, none of the IMBT should be modified: this C_1^* cluster becomes an immutable one, $C_1^* \Rightarrow C_1$. Parallel to the last synchronization message in C_1 a new, C_2^* cluster starts its operation, composed from exactly k new, mutable empty IMBT nodes.

To satisfy the filtering condition, from now on, first the keys stored in C_1 will be queried. Since the nodes are exactly the same in C_1 any node can be contacted. In case of the key is already stored in one of the intervals then the search operation stops and the $(key, value)$ pair will be dropped, due to duplication.

Otherwise the C_2^* will be queried. Again: any node can be contacted, since the IMBTs are almost the same, and due to the synchronization procedure introduced in *Synchronization Methods* the duplication will be filtered out, or detected at least. That is, the average cost of search operation is composed of a sub-search of an immutable IMBTs, and a sub-search of mutable IMBTs. However, the number of vertices in any IMBT inside C_2^* less than v_t, otherwise it would turn to be C_2. So, the framework can decide about a k_i key if it has been stored before or not roughly in average $2 \times log(v_t)$ comparison steps. The above described scenarios are visualized in Fig. 3.22 Let's see the

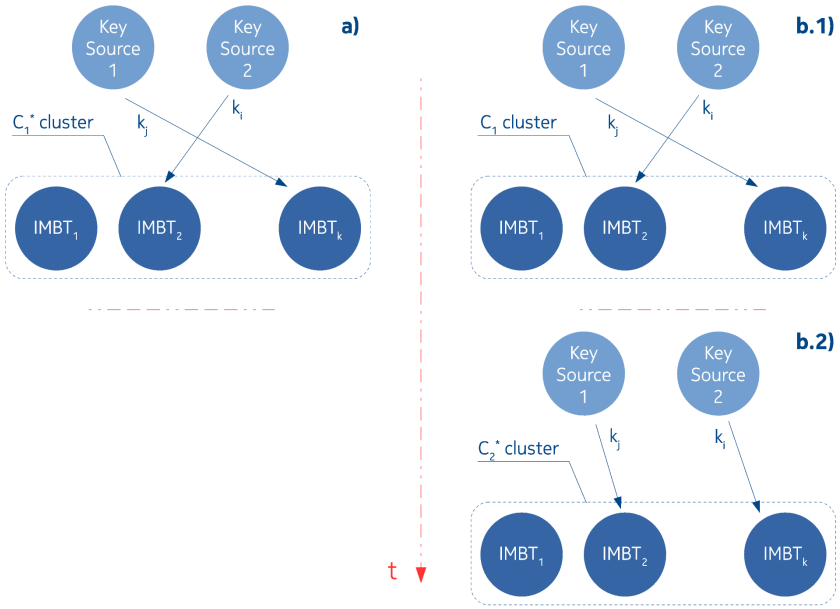

Figure 3.22: IMBT Cluster Based Space Scale Out a) Initial Cluster b) Duplicated Cluster b.1) First the Immutable C_i is queried b.2) Then the Mutable C_i^* is queried.

further step, when $C_2^* \Rightarrow C_2$ turns to full. In this case C_3^* will be initialized with exactly the same (k) number of IMBTs inside. Now, to be able to keep the average answer time under $2 \times log(v_t)$, we emit parallel search operations to C_1 and C_2. Since both of them contains exactly k IMBTs with exactly v_t nodes, but with completely disjoint intervals, the first answer will arrive within $log(v_t)$ comparisons time. Suppose that the key is a new

one: then we must turn to the C_3^*, in which any IMBT has less than v_t nodes (otherwise it would turn to be an immutable C_3). So, here the query also will be terminated within $log(v_t)$ comparison time. That is the framework can serve any key related query within in average $2 \times log(v_t)$ comparison steps, Fig. 3.23.

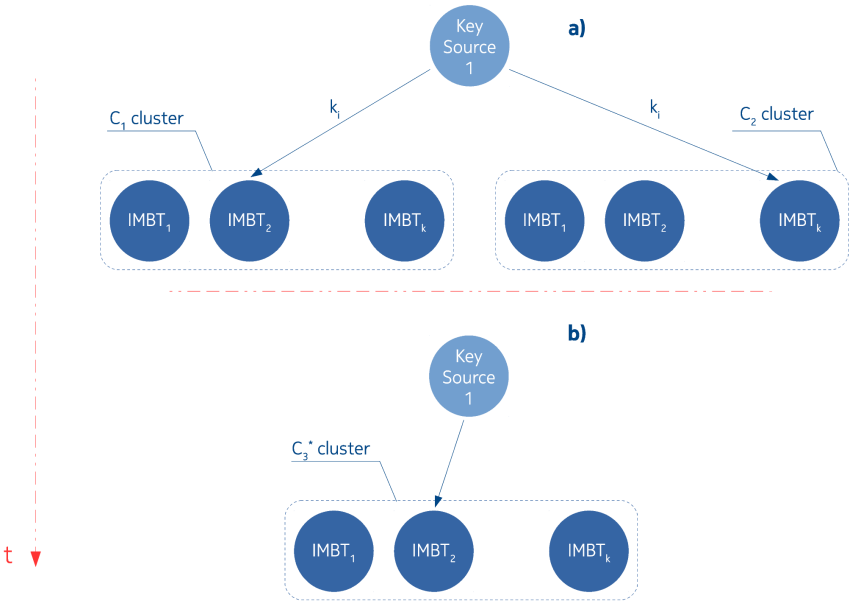

Figure 3.23: IMBT Cluster Based Space Scale Out a) Parallel Queries Against the Immutable IMBTs b) Then Query Against the Mutable IMBT.

Processing Performance Scaling

a) Scaling with Identical IMBTs

In case when the intensity of the incoming keys is increasing, then the increasing number of replicas is a possible way to keep the average response time under a certain limit, see Fig 3.24. The extension operation is a simple copy of any of the existing IMBT from the C_i. In the C_i^* an additional insertion into the synchronization ring is also required with the modification of two pointers. This action, along with given v_t, will not influence the rate of the $C_i^* \Rightarrow C_i$ transitions; that rate increases with the increase of the input load. The computing performance of the C_i clusters are increasing, but the parallel emitted messages as well. Additionally, this action increases the number of IMBTs in the C_i^* during synchronization, which increases the *transactional delay*. Therefore the applier must find the trade-off. In the opposite case the intensity of the streaming keys is decreasing. In this case as many IMBTs can be freed up in each and every C_i and the C_i^* as the remaining ones still able to filter the streams of keys. The only constraint is that minimum one instance must remain from all IMBTs.

b) Scaling with v_t of IMBTs

As we presented in the *Synchronization Methods* there is a strict connection between the number of vertices in the IMBT and the cost of merging operations (and serialization/deserialization as well). Therefore it is a way to theoretically increase the performance of the filter framework that we intentionally set v_t to lower value. This speeds up

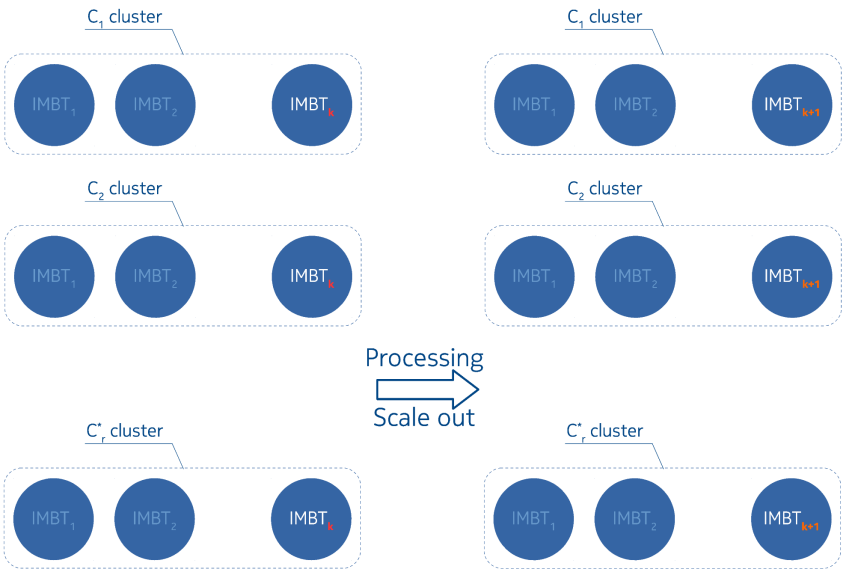

Figure 3.24: IMBT Increasing Number of Replicas per C_i to Handle the Increased Incoming Intensity.

both the parallel executed search operations and the synchronization process, since less vertices must be examined.

Let v_t is the power of two, let's say 2^p. Then, by setting $v_t = 2^{p+1}$, or $v_t = 2^{p-1}$ we can significantly change the both the duration of merging and serialization/deserialization operations; numerically with one comparison in average in case of every single key.

In case of decreasing v_t, the $C_i^* \Rightarrow C_i$ transition rates increasing, and as result the number of parallel emitted messages as well, which might be a bottleneck.

In the opposite case the increasing of v_t lead to less messages, but increasing the cost of search operation and both the synchronization and serialization/deserialization as well.

Therefore, the determination of v_t requires careful planning.

Additionally, wrongly chosen v_t can led to such permanent gaps in the IMBTs belong to different clusters, which would not be there with that high $C_i^* \Rightarrow C_i$ rate, or would not be there at all. Therefore we have to be convinced, via measuring, that the slicing is unavoidable and v_t is well determined.

To this end, we should measure the of that search operations in C_i which ends with "not found" on such leaves, which are not extremes (not the leftmost nor the rightmost leaf), however, still insertion is taken place in C_i^*. This may indicate that if a C_i would still act like a C_i^*, then key would be inserted into the tree. Therefore this insertion may lead to any kind of interval increasing, moreover vertex decreasing. That is, the whole filtering framework could perform more efficiently.

It is important to note that all the above mentioned methods could be applied not only to IMBTs, but traditional BSTs as well. What makes it more efficient is the fact that the number of synchronization messages can be significantly smaller, than with the traditional case. This is a kind of buffering effect of the application of intervals.

The section relates to *Thesis 4.2.5*.

Chapter 4

Conclusion - Theses

4.1 Theses Group - Lossless Data Compression

4.1.1 Thesis - VDE Compression Method

I worked out the LZW based VDE-LGD lossless data compression method, which fulfills that requirements against lossless compression methods, according to which by the application of the D decoding method on the A2=E(A1) data, where E is encoding compression method, then as a result the original, A1 input data is given back, that is:

$$A1 = D(A2),\ where\ A2 = E(A1),\ that\ is\ A1 = D(E(A1)). \tag{4.1}$$

The VDE-LGD compression method is actually a sort of extension of the dictionary based, lossless compression, the LZW. In case of LZW the compression gain come from the substitution of dictionary entries by their dictionary position, which can be significantly shorter than the entry itself 2.2, [IF-02].

In case of VDE I have associated another identifier, the so called index besides the position. The so called primary indices, which are associated to direct entries are differ from the values of their positions, except on the zero and the first positions. In case of LGD the values of primary indices growing according to the triangle numbers. This extension through the so called virtual indices, which fall between two primary indices allows us to uniquely identify the different concatenations of the primary entries. Thus, if instead of positions the indices are stored, then in case of favorable input pattern far longer entries can be substituted with numbers than in case of positions based substitutions. Such a virtual word is visible In Fig. 4.1.

During the dictionary building on the decoding side that condition is heavily used that the value of primary entries have to be identical with the triangle numbers.

4.1.2 Thesis - VDE Analysis

I have worked out a storage need intensity based model, which facilitates the determination of the compression ratio, time and space complexities, as the fingerprint function of the length and statistical characteristics of the input data. Through the metrics of R_b (- the

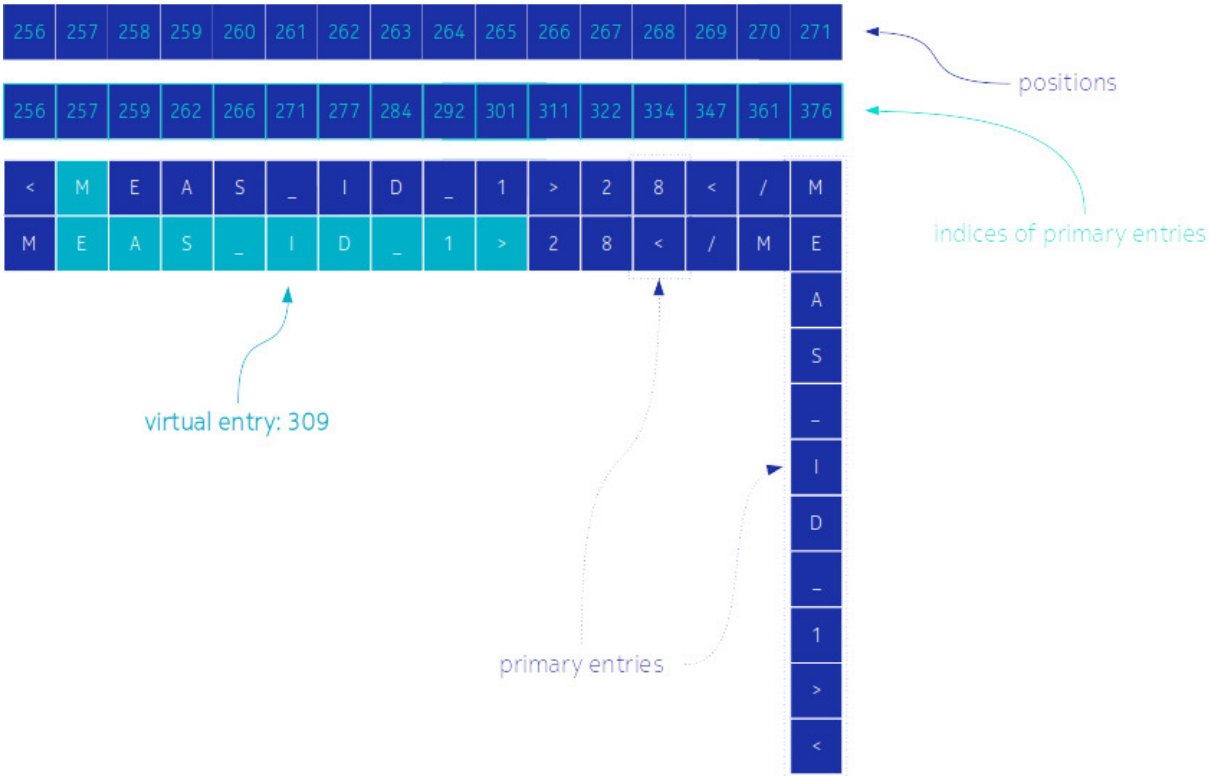

Figure 4.1: Virtual word example

required number of bits to represent the full dictionary) and f_m (- the number of final matches), introduced during the modeling, the VDE-LGD method gets comparable with LZW. 2.3 [IF-03, IF-08]

Based on the model I performed the comparisons in the following extreme cases out of the dependencies visible in Fig. 2.1

- the following asymptotic space complexities belongs to the most storage demanding input pattern: $S_{LZW}(p) = \mathrm{O}(p^2)$ in case of LZW, compared to the $S_{LGD}(p) = O(2^p)$ in case of VDE-LGD, where p refers to the positions of the individual entries;

- the least storage demanding input pattern in case of LZW is unequivocal if all the words stored with maximum q length: $S_{LZW}(q) = \sum_{i=1}^{q} V_b^{r,i+1}$. where V refers to variation, b is the cardinality of the alphabet, r in the upper index means that repetition is allowed and the number in the upper index refer to the length of the word over the alphabet. However, the determination of the least storage demanding input pattern of VDE-LGD is hard due to the presence of the implicit recursive dependencies. Therefore, in this case I only have proved a required condition, next to which the $S_{LGD}(q)$ remains asymptotic true, that is $S_{LGD}(q) \approx \sum_{i=1}^{q} V_b^{r,i+1}$.

- the compression ratio in the most storage demanding case is . Assuming finite dictionary size I proved that CR_{LGD} tends to Shannon limit more faster than CR_{LZW}. While, I proved that in the least space demanding case $CR_{LGD} \approx 2 * CR_{LZW}$.

- the encoding time complexity for given input length, n, both in case of LZW and LGD can be expressed with the $TLZW(n) = Tr(n)+Tcomp(n)+Tde(n)+Tins(n)+$

$Twr(n)$ formula. However, the LZW's T_r, T_{comp} and T_{de} operations are squared proportional to LGD's same operations. This fact clearly show that the memory resources has been changed to computation resources.

The related deductions and proves are available in section 2.3.1, 2.3.2 and 2.3.3, respectively.

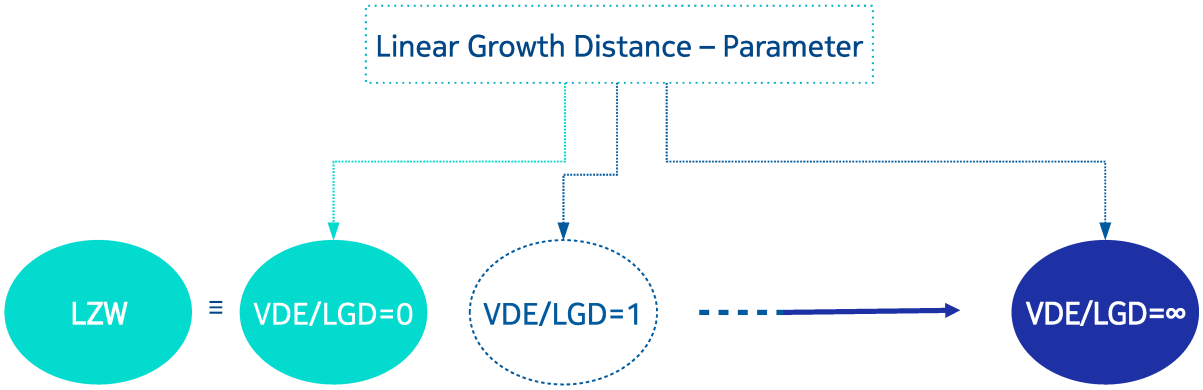

Figure 4.2: Parameterized VDE-LGD method, where LZW is identical to LGD=0 parameter.

4.2 Theses Group - Data Structures and Data Management

4.2.1 Thesis - Interval Merging Binary Tree

I have constructed a data structure, the Interval Merging Binary Tree (IMBT), which might be very efficient for duplication-free storage, in case of the payload can be identified with such individual keys, which resides close to each other in the key-space. I proved if the data structure can be significantly more efficient in case of favorable key distribution, than the already existing ones.

In case of the filtering of unique key identified data most of the data structures stores each of the identifiers one-by-one. Next to near real time processing this solution might be easily the bottleneck, even in case of the data structure is the hash-table, where average operation cost can be characterized with O(1). Namely, the precondition of fast operations is that the data or data structures resides in the memory. However, in such an enormous amount of data, like monitoring of a telecommunication network, the memory can become easily a bottleneck. To this end I have worked out the IMBT, which has a stochastically constant memory need (in Fig 4.3). Otherwise, its memory need and in accordance to this the searching operation times complexity is slowly growing.

The related deductions and proves are available in section 3.2.

4.2.2 Thesis - IMBT State Space

I have setup a mathematical model, with the help of which the cardinality of the different number of time complexities can be upper estimated. [IF-05]

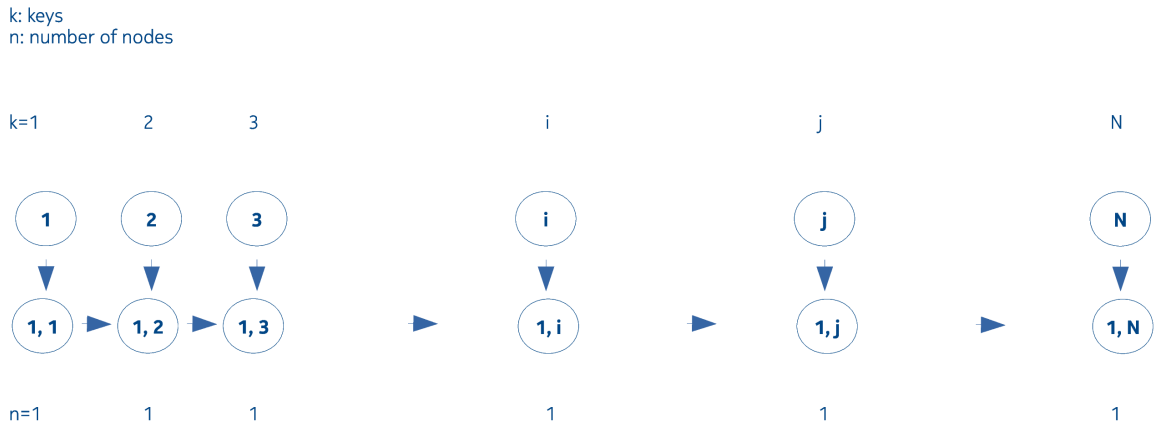

Figure 4.3: Interval Merging Binary Tree (IMBT) number of keys increasing and the interval evolving while the number of nodes is constant.

During the first part of the proof I have proved that number of different arrangements of the IMBT is identical with number of integer partitions of N. Some naive realizations are visible in Fig. 3.3,3.4, 3.5. Then I have determined the number of classes, which lead to different traversal weights, assuming a balanced tree. Finally, by the combination of the previous outcomes, I determined an upper estimation regarding the possible number of different time complexities next to given N.

The results are confirm that the efficiency of IMBT is highly influenced by the distribution of incoming keys, besides the balancing. Thus, it is reasonable perform statistical key distribution based investigations as well.

The related deductions and proves are available in section 3.3.

4.2.3 Thesis - IMBT Special Conditions

I have worked out a computation method, which based on contingency tables. By this method the average cost of search operation of the IMBT theoretically can be determined.

I have deducted and proved that next to given traversal the search performance of the IMBT is better compared to the traditional BST-s, until the $a \geq \sqrt[3]{N}$ *inequality holds, assuming that the a average interval length is distributed uniformly in the nodes of the tree.* [IF-06]

I have shown that if the conditions, which can be characterized with Bernstein-theorems[60] are hold, then the average length of the intervals in the tree are fluctuating around a constant expected value, next to the growing value of N. Otherwise, length of the intervals tends to ∞, next to finite contingency table, and the height of the IMBT is fluctuating around a constant value. This results an O(1) time complexity, in terms of search operations, independently from N, Fig. 4.4 (source [IF-06]).

In case of certain key distributions the height of the IMBT is fluctuating around a constant value, independently from the number of keys. The width of the blue stripe depends on the shuffling of the keys. This kind of key distributions lead to stationary space and time complexities.

The related deductions and proves are available in section 3.4.

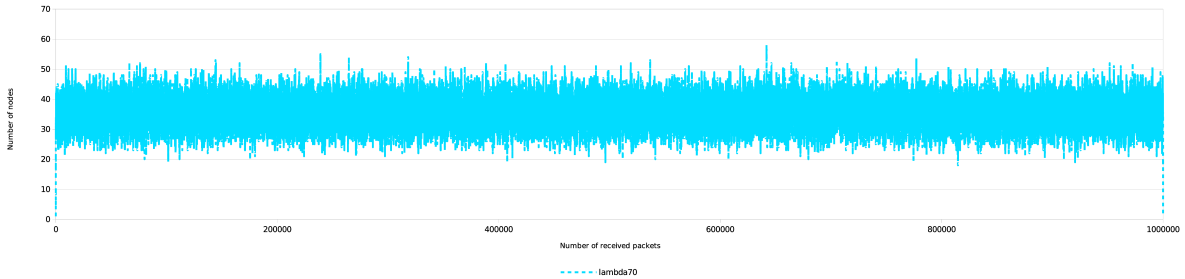

Figure 4.4: Balanced IMBT, temporary gaps only, O(1) time complexity. The width of the blue stripe depends on the shuffling of the keys.

4.2.4 Thesis - IMBT Matrix Representation and an Equilibrium Condition

I have introduced the matrix representation of the IMBT, with the help of which arbitrary key-distribution easily can be encoded. I gave a required condition of the existence of equilibrium points in case of series of geometric progressions.

The following formula expresses the required change of the b base as a function of N to maintain the constant number of nodes in the tree, next to the geometric progression based distribution of the subsequent interval lengths.

$$b_e(N) = x^{\frac{log_x(N)}{const.}} = x^{\frac{log_x(N)}{n}} \tag{4.2}$$

The related deductions and proves are available in section 3.5.

4.2.5 Thesis - IMBT in Distributed Environment

I have worked out the application of the IMBT in distributed environment, where synchronization between the individual nodes can be executed more effectively compared to other data structures, due to the buffering effect of the IMBT.

I have proved that by the application of proper scaling methods, where trees are organized into so called immutable groups the search operation cost is constant, independently from the number of keys. [IF-07]

The C_i clusters visible in Fig. 3.24 are immutable, that is why parallel queries are allowed. Since the height of the trees are pre-defined in this solution, the same log(h) + 1 cost is provided. Additionally, the height constraint is valid for the C* under change as well. That is, the search cost is surely less or equal than 2*(log(h)+1).

During the above introduced solution the number of parallel emitted messages might become the bottleneck. I am working on a solution, which might solve this problem with constant number of messages under certain circumstances, or with log_T number of messages, where $T >> 2$ in worst case. The results are promising, but are not published yet.

The related deductions and proves are available in section 3.6.

Chapter 5

Applicability of the Results

I have worked out both the compression method and the IMBT due to industrial need. Therefore the practical applicability is well-founded.

Moreover, the quality analysis of the VDE's worst case may lead to such the field of mathematical methods, from which I expect more abstract, more general results regarding the repetition-free strings (combinatorics on words, colored exponential trees).

I am working on a solution, which might solve this problem with constant number of messages under certain circumstances, or with log_T number of messages, where $T >> 2$ in worst case. The results are promising.

References

[1] Amazon AWS, https://aws.amazon.com/, last visited 2021-02-20

[2] Dean, J.; Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, http://static.googleusercontent.com/media/research.google.com/es/us/archive/mapreduce-osdi04.pdf, last visited 2021-02-20

[3] GFS Architecture, https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf, last visited 2021-02-20

[4] HDFS Architecture, https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html, last visited 2021-02-20

[5] MAPR (now part of HP) https://www.hpe.com/us/en/software/data-fabric.html, last visited 2021-02-20

[6] STORM - A distributed realtime computation system, http://storm.apache.org/documentation/Home.html , last visited 2021-02-20

[7] ZeroMQ messaging system https://zeromq.org/, last visited: 2021-02-20

[8] Netty messaging system https://netty.io/, last visited: 2021-02-20

[9] RabbitMQ messaging https://www.rabbitmq.com/, last visited: 2021-02-20

[10] Kafka messaging https://kafka.apache.org/, last visited: 2021-02-20

[11] Millwheel https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/41378.pdf , last visited 2021-02-20

[12] Samza https://samza.apache.org/, last visited 2021-02-20

[13] Spark https://spark.apache.org/, last visited 2021-02-20

[14] HBase https://hbase.apache.org/, last visited 2021-02-20

[15] Hive https://hive.apache.org/, last visited 2021-02-20

[16] Cassandra https://cassandra.apache.org/, last visited 2021-02-20

[17] CouchDB https://couchdb.apache.org/, last visited 2021-02-20

[18] VoltDB https://www.voltdb.com/, last visited 2021-02-20

[19] Zookeeper https://zookeeper.apache.org/, last visited 2021-02-20

[20] Ganglia monitoring system http://ganglia.sourceforge.net/, last visited: 2021-02-20

[21] Lambda architecture http://lambda-architecture.net/, last visited: 2021-02-20

[22] Java Collections Frameworks, http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html , last visited 2021-02-20

[23] Knuth, Donald (1997). "6.2.2: Binary Tree Searching". The Art of Computer Programming. 3: "Sorting and Searching" (3rd ed.). Addison-Wesley. pp. 426–458. ISBN 0-201-89685-0.

[24] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill, (2009). ISBN:0-262-03384-4

[25] Adelson-Velsky, G., Landis, E.: Organization and maintenance of large ordered indexes, Acta Informatica, Volume 1, Issue 3, pp. 173-189, (1972). DOI:10.1007/BF00288683

[26] Adelson-Velsky, G., Landis, E.: An algorithm for the organization of information, Proceedings of the USSR Academy of Sciences, Volume 146 , pp. 263-266 (1962).

[27] AVL Tree - Wiki, https://en.wikipedia.org/wiki/AVL-tree , last visited 2021-02-20

[28] Sedgewick, R.: Algorithms 1st edition, Addison-Wesley 1983, ISBN 0-201-06672-6.

[29] Bayer, R.: Symmetric binary B-Trees: Data structure and maintenance algorithms, Acta Informatica, Volume 1, Issue 4, pp. 290-306, (1972). DOI:10.1007/BF00289509

[30] B-Tree - Wiki, https://en.wikipedia.org/wiki/B-tree, last visited 2021-02-20

[31] Bayer, R.: Symmetric binary B-Trees: Data structure and maintenance algorithms, Acta Informatica, Volume 1, Issue 4, pp. 290-306, (1972). DOI:10.1007/BF00289509

[32] Red-black tree - Wiki, https://en.wikipedia.org/wiki/Red-black_tree, last visited 2017-03-29

[33] Paul E. Black, "(a,b)-tree", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed. 6 October 2004. (accessed 2021-03-22) Available from: https://www.nist.gov/dads/HTML/abtree.html

[34] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O.: Interval Trees, Computational Geometry, Second Revised Edition. Springer-Verlag, Section 10.1, pp. 212-217 (2000).

[35] Interval Tree https://en.wikipedia.org/wiki/Interval_tree, last visited 2021-02-20

[36] Bentley, J. L., Ottmann, T. A.: Algorithms for reporting and counting geometric intersections, IEEE Transactions on Computers, C28 (9), pp. 643-647, (1979). DOI:10.1109/TC.1979.1675432

[37] Bentley, J. L., Ottmann, T. A.: Algorithms for reporting and counting geometric intersections, IEEE Transactions on Computers, C–28 (9), pp. 643-647, (1979). DOI:10.1109/TC.1979.1675432

[38] Pfaff, B.: Performance analysis of BSTs in system software, ACM SIGMETRICS '04, Volume 32 Issue 1, pp. 410-422, (2004). ISBN:1-58113-873-3

[39] Bloom, B. H.: Space/time trade-offs in hash coding with allowable errors, Communications of the ACM, Volume 13 Issue 7, pp 422-426, New York, NY, USA, July (1970).

[40] Tom White: Hadoop: The definitive Guide, 2012 O'REILLY, ISBN: 978-1-449-31152-0-1327616795

[41] Jiang Liu; Bing Li; Meina Song: THE optimization of HDFS based on small files, Broadband Network and Multimedia Technology (IC-BNMT), 2010 3rd IEEE International Conference on, pp 913 - 915, 2010

[42] Zhang, Yang; Liu, Dan: Improving the Efficiency of Storing for Small Files in HDFS, Computer Science & Service System (CSSS), 2012 International Conference on, pp 2239 - 2242, 2012

[43] History of Lossless Data Compression Algorithms, http://ethw.org/History_of_Lossless_Data_Compression_Algorithms, last visited 2021-02-20

[44] Salomon, D., Motta, G.: Handbook of Data Compression, 5th edition London, England: Springer-Verlag, 2010, pp. 377-378. David Salomon, Giovanni Motta: Handbook of Data Compression, 5th edition London, England: Springer-Verlag, 2010, pp. 378-379.

[45] Triangular Number, http://mathworld.wolfram.com/TriangularNumber.html, last visited 2021-02-20

[46] Lempel, A., Ziv, J.: On the Complexity of Finite Sequences, IEEE Transactions on Information Theory, Volume 22 Issue 1, pp. 75 - 81, Jan 1976

[47] Ziv, J.: A constrained-dictionary version of LZ78 asymptotically achieves the finite-state compressibility with a distortion measure, IEEE Information Theory Workshop (ITW), pp. 1-4, 2015, Jerusalem, Israel

[48] Welch, T.: A Technique for High-Performance Data Compression, IEEE Computer Society Journal Volume 17 Issue 6, pp 8 - 19, June 1984

[49] Ziv, J.: A constrained-dictionary version of LZ78 asymptotically achieves the finite-state compressibility with a distortion measure, IEEE Information Theory Workshop (ITW), pp. 1 - 4, 2015, Jerusalem, Israel

[50] Shields, C.: Performance of LZ algorithms on individual sequences, IEEE Transactions on Information Theory, Volume 45 Issue 4, pp. 1283-1288, May 1999

[51] Nishad PM, Dr. R. Manicka Chezian: Behavioral Study of Data Structures on Lempel Ziv Welch (LZW) Data Compression Algorithm and ITS Computational Complexity, Intelligent Computing Applications (ICICA), 2014 International Conference on, pp. 268-274, March 2014

[52] Google Brotli https://en.wikipedia.org/wiki/Brotli, last visited 2021-02-20

[53] Google Snappy https://google.github.io/snappy/, last visited 2021-02-20

[54] Facebook Zstandard https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/, last visited 2021-02-20

[55] Facebook Zstandard https://en.wikipedia.org/wiki/Zstandard, last visited 2021-02-20

[56] Hardy, G.H., Ramanujan, S.: Asymptotic Formulae in Combinatory Analysis, Proceedings of the London Mathematical Society, 1918

[57] Bóna, M.: A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory. pp. 145-164, World Scientific Publishing, 2002 ISBN 981-02-4900-4.

[58] Cayley, A.: A Theorem on Trees. Quarterly Journal of Pure and Applied Mathematics 23, pp. 376-378, 1889

[59] Barvionk, A.: Enumerating Contingency Tables via Random Permanents, Combinatorics, Probability and Computing, Volume 17, pp. 1-19, 2008 DOI:10.1017/S0963548307008668

[60] Barvinok, A., Luria, A., Samorodnitsky, A., Yong, A.: An approximation algorithm for counting contingency tables, Random Structures Algorithms 37 (2010), no. 1, pp. 25-66, 2010 DOI:10.1002/rsa.20301 arXiv:0803.3948

[61] Stoica, I.; Morris, R.; Liben-Nowell, D.; Karger, D. R.; Kaashoek, M. F.; Dabek, F.; Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. IEEE/ACM Trans. Netw. 2003, 11(1), 17–32. DOI:10.1145/964723.383071

[62] Lauritzen, S.L. Lectures on Contingency Tables, 2002, Electronic edition, Aalborg University. Available online:http://www.stats.ox.ac.uk/~steffen/papers/cont.pdf, (accessed on 28 November 2019)

[63] Meyn, S.P. Tweedie, R.L. Markov Chains and Stochastic Stability. Springer: London, UK, 2012. ISBN9781447132677

[64] Bernstein, S.N. Theory of Probabilities. Moskva, Leningrad, 1946.

[65] FLINK - A framework and distributed processing engine for stateful computations over unbounded and bounded data streams, https://flink.apache.org/, last visited 2020-02-13

[66] Neuman, B.: Scale in Distributed Systems, Readings in Distributed Computing Systems, pages 463-489. IEEE Computer Society Press, Los Alamitos, CA., 1994

[67] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G.: Understanding Replication in Databases and Distributed Systems. In 20th International Conference on Distributed Computing Systems, pages 264-274, Taipei, Taiwan, Apr. 2000. IEEE. DOI: 10.1109/ICDCS.2000.840959

[68] Merkle, R., C.: "A Digital Signature Based on a Conventional Encryption Function". Advances in Cryptology — CRYPTO '87. Lecture Notes in Computer Science. 293. 1987 pp. 369-378 DOI: 10.1007/3-540-48184-2_32

[69] O'Neil, P., Cheng, E., Gawlick, D., O'Neil, E.: The Log-Structured Merge-Tree (LSM-Tree). Acta Informatica (1996) 33: 351. DOI: 10.1007/s002360050048.

[70] Sears, R., Ramakrishnan, R.: bLSM: A General Purpose Log Structured Merge Tree, SIGMOD '12, Scottsdale, Arizona, USA, May 20-24, 2012, pp. 217-228, DOI: 10.1145/2213836.2213862

[71] Lawder, J. and King, P.: Querying Multi-dimensional Data Indexed Using Hilbert Space-Filling Curve. ACM Sigmod Record, 30(1):19-24, Mar. 2000. DOI: 10.1145/373626.373678

[72] Vogels W.: Eventually consistent. Communications of the ACM, 52(1):40–44, Jan. 2009.

[73] Minsky, Y., Trachtenberg, A., Zippel, R.: Set Reconciliation with Nearly Optimal Communication Complexity, IEEE Transactions on Information Theory, Volume: 49, Issue: 9, Sept. 2003, pp 2213-2218 DOI: 10.1109/TIT.2003.815784

[74] Chen, D., Konrad, C., Yi, K., Yu, W., Zhang, Q.: Robust Set Reconciliation, SIGMOD '14 Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp 135-146 DOI: 10.1145/2588555.2610528

[75] Mehta, D., P., Sahni, S.: Handbook Of Data Structures And Applications, Chapman and Hall/CRC, 2004 ISBN:1584884355

[IF-01] Finta, I.; Farkas, L.; Sergyán, Sz.; Szénási, S.: Buffering Strategies in HDFS Environment with STORM framework, IEEE 16th International Symposium on Computational Intelligence and Informatics, 19–21 November, 2015, Budapest, Hungary

[IF-02] Finta, I.; Farkas, L.; Sergyán, Sz.; Szénási, S.: A Method for Virtual Extension of LZW Compression Dictionary, Innovations in Clouds, Internet and Networks (ICIN), 19th IEEE International Conference on, pp 184 - 188, 2016, Paris

[IF-03] Finta, I.; Farkas, L.; Sergyán, Sz.; Szénási, S.: Transient analysis of virtual dictionary extension compression method, IEEE 17th International Symposium on Computational Intelligence and Informatics (CINTI), pp. 67-74, 17-19 Nov. 2016, Budapest, Hungary DOI: 10.1109/CINTI.2016.7846381

[IF-04] Finta, I.; Farkas, L.; Sergyán, Sz.; Szénási, S.: Interval Merging Binary Tree, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017 DOI:10.1007/978-3-319-65482-9

[IF-05] Finta, I.; Szénási, S.: State-space Analysis of the Interval Merging Binary Tree. Acta Polytech. Hung. 2019 16(5), pp 71–85. DOI: 10.12700/APH.16.5.2019.5.5

[IF-06] Finta, I.; Szénási, S.; Farkas, L.: Input Pattern Classification Based on the Markov Property of the IMBT with Related Equations and Contingency Tables. Entropy 2020, 22, 245. https://doi.org/10.3390/e22020245

[IF-07] Finta, I.; Szénási, S.; Farkas, L.: Data Structure for Packet De-Duplication in Distributed Environments, 2020 IEEE Sixth International Conference on Big Data Computing Service and Applications, Oxford, United Kingdom, ISBN: 978-1-7281-7022-0

[IF-08] Finta, I.; Szénási, S.; Farkas, L.: Quantity Analysis on the Chaining of Repetition-free Words Considering the VDE Composition Rule, IEEE 15th International Symposium on Applied Computational Intelligence and Informatics, SACI2021

[IF-09] Finta, I.; Szénási, S.; Farkas, L.: Quantifying the Performance of Interval Merging Binary Trees using a Matrix Representation, The 2021 World Congress in Computer Science, Computer Engineering, & Applied Computing - CSCE '21, Las Vegas, USA, 2021

[IF-10] Horvath, I.; Finta, I.; Kovacs, F.; Meszaros, M.; Molontay, R.; and Varga, K.: Markovian queue with garbage collection. In: Thomas N., Forshaw M. (eds) Analytical and Stochastic Modelling Techniques and Applications. ASMTA 2017. Lecture Notes in Computer Science, vol 10378. Springer, Cham. https://doi.org/10.1007/978-3-319-61428-1_8

[IF-11] Finta, I.; Élias, G.; Illés, J.: Packet Loss and Duplication Handling in Stream Processing Environment. In Proceedings of the CINTI 2018, Budapest, Hungary, 21–22 November 2018. DOI:10.1007/978-3-319-65482-9

APPENDICES

Appendix A

VDE Pseudo Code

A.1 Encoding - Java Like

```
private InputStream is;
private OutputStreamWriter os_stat;
private OutputStream os_stat_orig;
private final int baseLength = 256;
private int extendedLength = 256;

private int valueToWrite;
private int dictAbsPosition;
private HashMap<String, Integer> dictionary;
private String[] dictionaryIndex;
private StringBuffer sb = new StringBuffer(this.extendedLength);
private String initChars = "";
private HashSet<Integer> reservedIntegers = new
↪  HashSet<Integer>(initChars.length());

private boolean firstMatchBool = true;
private int firstMatchPos = 0;
private int wordPos = 0;
private int tempPos = 0;
private String nextWord = null;
private StringBuffer buffer = new StringBuffer(this.extendedLength);

private String destination = "";

private CompressionWriter cw;

private boolean logging = false;
int lenght_length = 5;
StringBuffer length_base = new StringBuffer ("00000");
String length_final = "";
```

```
public void encode (){
    int c = 0;
    this.dictAbsPosition = this.baseLength;
    String character = "";
    int internal_value_to_write = 0;
    if (this.sb.length() > 0)
        this.sb.delete(0, (this.sb.length() - 1));
    while ((c != -1) || (this.buffer.length() >= 0) || (this.sb.length()
    ↪  >= 0)) {
        if (this.firstMatchBool) {
            if ((c  != -1) && (this.buffer.length() < 1)) {
                if ((c = this.is.read()) != -1) {
                    character = new String(Character.toChars(c));
                    this.buffer.append(character);
                    os_stat_orig.write(c);
                } else {
                    break;
                }
            }
            if (this.dictionary.containsKey(this.sb +
            ↪  this.buffer.substring(0, 1))) {
                this.sb.append(this.buffer.substring(0, 1));
                this.buffer.delete(0, 1);
            } else {
                this.firstMatchPos = this.dictionary.get(new
                ↪  String(this.sb));
                if (this.sb.length() > 1) {
                    this.wordPos++;
                    if (((this.firstMatchPos + this.wordPos) <
                    ↪  this.dictionaryIndex.length) &&

                            ↪  !("0".equals(this.dictionaryIndex[this.firstMatchPos
                            ↪  + this.wordPos]))) {
                        this.firstMatchBool = false;
                        this.nextWord =
                        ↪  this.dictionaryIndex[this.firstMatchPos +
                        ↪  this.wordPos].substring(1);
                    } else {
                        //no more entry, write out and update dictionary
                        this.wordPos = 0;
                        internal_value_to_write = this.valueToWrite();
                        cw.write(internal_value_to_write);
                        this.dictionaryOverflowHandling();
                        this.sb.delete(0, this.sb.length());
                        this.sb.append(this.buffer.substring(0, 1));
                        this.buffer.delete(0, 1);
                    }
                } else {
```

```
                internal_value_to_write = this.valueToWrite();
                cw.write(internal_value_to_write);
                this.dictionaryOverflowHandling();
                this.sb.delete(0, this.sb.length());
                this.sb.append(this.buffer.substring(0, 1));
                this.buffer.delete(0, 1);
            }
        }
    } else {
        if ((c  != -1) && ((this.buffer.length() - this.tempPos) <
        ↪  1)) {
            if ((c = this.is.read()) != -1) {
                character = new String(Character.toChars(c));
                this.buffer.append(character);
                os_stat_orig.write(c);
            } else {
                continue;
            }
        }
        // Step over the word from dictionary
        if (this.tempPos < this.nextWord.length()) {
            if (((this.buffer.length() - 1) >= this.tempPos) &&
            ↪  (this.nextWord.charAt(this.tempPos) ==
                    this.buffer.charAt(this.tempPos))) {
                    this.tempPos++;
                } else {
                    //write out and go to first match with characters
                    ↪  buffered
                    this.wordPos--;
                    internal_value_to_write = this.valueToWrite();
                    cw.write(internal_value_to_write);
                    this.dictionaryOverflowHandling();
                    this.sb.delete(0, this.sb.length());
                    if (this.buffer.length() > 0) {
                        this.sb.append(this.buffer.substring(0, 1));
                        this.buffer.delete(0, 1);
                    }
                    if (c != -1) {
                        this.tempPos = 0;
                        this.wordPos = 0;
                        this.firstMatchBool = true;
                    } else {
                        break;
                    }
                }
        // Search for new entry in the dictionary
        } else {
            this.sb.append(nextWord);
```

```
                this.buffer.delete(0, this.tempPos);
                this.wordPos++;
                if (((this.firstMatchPos + this.wordPos) <
                ↪  this.dictionaryIndex.length) &&

                        ↪  !("0".equals(this.dictionaryIndex[this.firstMatchPos
                        ↪  + this.wordPos]))) {
                    this.nextWord =
                    ↪  this.dictionaryIndex[this.firstMatchPos +
                    ↪  this.wordPos].substring(1);
                } else {
                    //no more entry, write out and update dictionary
                    this.wordPos--;
                    internal_value_to_write = this.valueToWrite();
                    cw.write(internal_value_to_write);
                    this.dictionaryOverflowHandling();
                    this.sb.delete(0, this.sb.length());
                    this.sb.append(this.buffer.substring(0, 1));
                    this.buffer.delete(0, 1);
                    this.wordPos = 0;
                    this.firstMatchBool = true;
                }
                this.tempPos = 0;
            }
        }
    }
}
```

A.2 Decoding - Java Like

```
private OutputStream os;
private final int baseLength = 256;
private final int extendedLength = 256;

private CompressionReader cr;

private String[] dictionary;
private int dictAbsPosition;
private double position;
private double flooredPosition;
private double numberOfHops = 0;
private String derivedWord = "";

private String initChars = "ab";
```

```
private HashSet<Integer> reservedIntegers = new
↪  HashSet<Integer>(initChars.length());

public void decode () {
    this.dictAbsPosition = this.baseLength;
    int j = 0;
    String sk = "";
    char character;

    j = this.cr.decode();
    for (int l = 0; l < this.dictionary[j].length(); l++) {
        this.os.write(this.dictionary[j].charAt(l));
    }
    sk = this.dictionary[j];
    j = 0;
    while ((j = this.cr.decode()) != -1) {
        if (j < this.baseLength) {
            this.position = j;
            this.flooredPosition = j;
        } else {
            this.position = ((Math.sqrt((1 + 8*(j - this.baseLength))) -
            ↪  1) / 2) + this.baseLength;
            this.flooredPosition = Math.floor(this.position);
        }
        // If position is an "integer" it is a primary entry otherwise it
        ↪  is a derivative
        if (this.position != this.flooredPosition) {
            this.numberOfHops = j - this.baseLength -
            ↪  (((this.flooredPosition - this.baseLength) *
            ↪  ((this.flooredPosition - this.baseLength) + 1)) / 2);
            if (this.dictionary[(int)this.flooredPosition] != null) {
                for (int l = (int)(this.flooredPosition -
                ↪  this.numberOfHops); l <= (int)this.flooredPosition;
                ↪  l++) {
                    int m = 0;
                    if (l != (int)(this.flooredPosition -
                    ↪  this.numberOfHops)) {
                        m = 1;
                    }
                    for (; m < this.dictionary[l].length(); m++) {
                        this.os.write(this.dictionary[l].charAt(m));
                        this.derivedWord = this.derivedWord +
                        ↪  this.dictionary[l].charAt(m);
                    }
                }
                character = this.derivedWord.charAt(0);
                this.dictionary[this.dictAbsPosition] = sk + character;
                sk = new String(this.derivedWord);
```

```
                this.derivedWord = "";
            } else {
                character = this.dictionary[(int)(this.flooredPosition -
                ↪  this.numberOfHops)].charAt(0);
                this.dictionary[this.dictAbsPosition] = sk + character;
                for (int l = (int)(this.flooredPosition -
                ↪  this.numberOfHops); l <= (int)this.flooredPosition;
                ↪  l++) {
                    int m = 0;
                    if (l != (int)(this.flooredPosition -
                    ↪  this.numberOfHops)) {
                        m = 1;
                    }
                    for (; m < this.dictionary[l].length(); m++) {
                        this.os.write(this.dictionary[l].charAt(m));
                        this.derivedWord = this.derivedWord +
                        ↪  this.dictionary[l].charAt(m);
                    }
                }
                sk = new String(this.derivedWord);
                this.derivedWord = "";
            }
        } else {
            if (this.dictionary[(int)this.flooredPosition] != null) {
                for (int l = 0; l <
                ↪  this.dictionary[(int)this.flooredPosition].length();
                ↪  l++) {

                    ↪  this.os.write(this.dictionary[(int)this.flooredPosition].charA
                }
                character =
                ↪  this.dictionary[(int)this.flooredPosition].charAt(0);
                this.dictionary[this.dictAbsPosition] = sk + character;
            } else {
                character = sk.charAt(0);
                this.dictionary[this.dictAbsPosition] = sk + character;
                for (int l = 0; l <
                ↪  this.dictionary[this.dictAbsPosition].length(); l++)
                ↪  {

                    ↪  this.os.write(this.dictionary[this.dictAbsPosition].charAt(l)
                }
            }
            sk = this.dictionary[(int)this.flooredPosition];
        }
        this.dictionaryOverflowHandling();
    }
}
```

Appendix B

IMBT Search, Insert and Remove pseudo codes

B.1 Search

```
/*****************************************************************
                          SEARCH PSEUDO
*****************************************************************/

function node SEARCH (tree T, key K) {
 node node;
 if(T.root == NULL) {
  /*We are ready the tree is empty*/
  return NULL;
 } else {
  node = T.root;
  while (node != NULL) {
   if (K < node.domain_Left_Value) {
    node = node.pointer_to_Left_Child;
   } elseif (K > node.domain_Left_Value) {
    node = node.pointer_to_Right_Child;
   } else {
    return node;
   }
  } // end of while
  return NULL;
 }
}
```

B.2 Insert

```
/*****************************************************************
                          INSERT PSEUDO
*****************************************************************/

function INSERT (tree T, key K) {
 if(T.root == NULL) {
  /*Generate a root node with boundaries/domain values (K, K);*/
  T.root = new node(K, K);
 } else {
  /*Search for the place of the new key K, with the following rules*/
  node = T.root;
  while (node != NULL) {
   if (node.domain_Left_Value <= K <=  node.domain_Rigth_Value) {
    drop K; /*already stored*/
    node = NULL;
   } else {
    /* Left Branch */
    if (K < node.domain_Left_Value){
     if ((node.domain_Left_Value { K) > 1) {
      if (node.pointer_to_Left_Child != NULL) {
       node = node.pointer_to_Left_Child;
      } else {
       node.pointer_to_Left_Child = new node(K, K);
       node = NULL;
      }
     /* checkMerging will be performed */
     } else {
      node.domain_Left_Value = K;
      checkMerging(node, L);
      node = NULL;
     }
    /* Right Branch */
    } else {
     if ((K - node.domain_Right_Value) > 1) {
      if (node.pointer_to_Right_Child != NULL) {
       node = node.pointer_to_Right_Child;
      } else {
       node.pointer_to_Right_Child = new node(K, K);
       node = NULL;
      }
     /* checkMerging will be performed */
     } else {
      node.domain_Right_Value = K;
      checkMerging(node, R);
      node = NULL;
     }
    }
   }
```

```
  } //end of while
 }
}

function checkMerging(node N, direction D) {
 node origNode = N;
 node subNode;
 boolean found = false;
 if (D == L) {
  if (origNode.pointer_to_Left_Child != NULL) {
   subNode = origNode.pointer_to_Left_Child;
  } else {
   found = true;
  }
 } else {
  if (origNode.pointer_to_Right_Child != NULL) {
   subNode = origNode.pointer_to_Right_Child;
  } else {
   found = true;
  }
 }

 while (!found) {
  /* Left Branch */
  if (D == L) {
   /* Search for highest smaller node */
   if (subNode.pointer_to_Right_Child != NULL) {
    subNode = subNode.pointer_to_Right_Child;
   } else {
    if ((origNode.domain_Left_Value { subNode.domain_Right_Value) == 1) {
     merging(origNode, subNode, L);
    }
    found = true;
   }
  /* Right Branch */
  } else {
   /* Search for smallest higher node */
   if (subNode.pointer_to_Left_Child != NULL) {
    subNode = subNode.pointer_to_Left_Child;
   } else {
    if ((subNode.domain_Rigt_Value { origNode.domain_Right_Value) == 1) {
     merging(origNode, subNode, R);
    }
    found = true;
   }
  }
 }
}
```

```
function merging (node origNode, node subNode, direction D) {
 /* Left Branch */
 if (D == L) {
  /* subNode is directly child of the origNode */
  if (origNode.pointer_to_Left_Child == subNode) {
    origNode.domain_Left_Value = subNode.domain_Left_Value;
   /* node is not a leaf */
   if (subNode.pointer_to_Left_Child != NULL) {
    origNode.pointer_to_Left_Child =  subNode.pointer_to_Left_Child;
   /* node is a leaf */
   } else {
    origNode.pointer_to_Left_Child = NULL;
   }
  /* subNode has not direct connection to origNode */
  } else {
   origNode.domain_Left_Value = subNode.domain_Left_Value;
   /* node is not a leaf */
   if (subNode.pointer_to_Left_Child != NULL) {
    subNode.parent.pointer_to_Left_Child =  subNode.pointer_to_Left_Child;
   /* node is a leaf */
   } else {
    subNode.parent.pointer_to_Left_Child = NULL;
   }
  }
 /* Right Branch */
 } else {
  /* Same as Left Branch with opposite directions */
 }
}
```

B.3 Remove

```
/*****************************************************************
                          REMOVE PSEUDO
*****************************************************************/

function REMOVE (tree T, key K) {
 node node;
 node subNode;
 if(T.root == NULL) {
  /*We are ready the tree is empty*/
 } else {
  /*Search for key K*/
  node = T.root;
```

```
while (node != NULL) {
 /* Left Branch */
 if (K < node.domain_Left_Value) {
  if (node.pointer_to_Left_Child != NULL) {
   node = node.pointer_to_Left_Child;
  } else {
   /* the Tree does not contain the Key */
   node = NULL;
  }
 /* Right Branch */
 } elseif (K > node.domain_Left_Value) {
  if (node.pointer_to_Right_Child != NULL) {
   node = node.pointer_to_Right_Child;
  } else {
   /* the Tree does not contain the Key */
   node = NULL;
  }
 } else {
  /* Node which contains the K key has been found */
  if (node = T.root) {
   /* The root contains only 1 Key */
   if (node.domain_Left_Value == node.domain_Rigth_Value) {
    if (node.pointer_to_Right_Child != NULL) {
     if (node.pointer_to_Left_Child != NULL) {
       subNode = searchForSmallestSubNode(node.pointer_to_Right_Child);
       subNode.pointer_to_Left_Child = node.pointer_to_Left_Child;
     }
     T.root = node.pointer_to_Right_Child;
    } else {
     if (node.pointer_to_Left_Child != NULL) {
      T.root = node.pointer_to_Left_Child;
     } else {
      T.root = NULL;
     }
    }
   } else {
    /* node covers only 2 elements */
    if ((node.domain_Rigth_Value { node.domain_Left_Value) == 1) {
     if (K == node.domain_Left_Value) {
      node.domain_Left_Value = node.domain_Right_Value;
     } else {
      node.domain_Right_Value = node.domain_Left_Value;
     }
    /* node covers more than 2 elements */
    } else {
     /* K is one of the boundary value */
     if (K == node.domain_Left_Value) {
       node.domain_Left_Value = K + 1;
```

```
   } elseif (K == node.domain_Right_Value) {
     node.domain_Right_Value = K { 1;
   /* K is one element from the middle of the domain */
   } else {
    subNode = new node(T.root.domain_Left_Value, K-1);
    subNode.pointer_to_Left_Child = T.root.pointer_to_Left_Child;
    T.root.domain_Left_Value = K + 1;
    T.root.pointer_to_Left_Child = subNode;
   }
  }
 }
/* node != T.root */
} else {
 /* node is a left-node */
 if (node == node.parent.pointer_to_Left_Child) {
  /* The node contains only 1 Key
  if (node.domain_Left_Value == node.domain_Rigth_Value) {
   if (node.pointer_to_Right_Child != NULL) {
    if (node.pointer_to_Left_Child != NULL) {
      subNode = searchForSmallestSubNode(node.pointer_to_Right_Child);
      subNode.pointer_to_Left_Child = node.pointer_to_Left_Child;
    }
    node.parent.pointer_to_Left_Child = node.pointer_to_Right_Child;
   } else {
    if (node.pointer_to_Left_Child != NULL) {
     node.parent.pointer_to_Left_Child = node.pointer_to_Left_Child;
    } else {
     node.parent.pointer_to_Left_Child = NULL;
    }
   }
  } else {
   /* node covers only 2 elements */
   if ((node.domain_Rigth_Value { node.domain_Left_Value) == 1) {
    if (K == node.domain_Left_Value) {
     node.domain_Left_Value = node.domain_Right_Value;
    } else {
     node.domain_Right_Value = node.domain_Left_Value;
    }
   /* node covers more than 2 elements */
   } else {
    /* K is one of the boundary value */
    if (K == node.domain_Left_Value) {
      node.domain_Left_Value = K + 1;
    } elseif (K == node.domain_Right_Value) {
      node.domain_Right_Value = K { 1;
    /* K is one element from the middle of the domain */
    } else {
     subNode = new node(node.domain_Left_Value, K-1);
```

```
          subNode.pointer_to_Left_Child = node.pointer_to_Left_Child;
          node.domain_Left_Value = K + 1;
          node.pointer_to_Left_Child = subNode;
          subNode.parent = node;
         }
        }
      /* node is a right-node */
      } else {
       /* regarding the parent everything is the opposite */
      }
     }
     node = NULL;
    }
   } //end of while
  }
 }
```