TECHNISCHE
UNIVERSITÄT
DRESDEN

Resiliency Mechanisms for In-Memory Column Stores

Dissertation

zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

eingereicht von
Dipl.-Inf. Till Kolditz
geboren am 10. September 1985 in Berlin-Mitte

Gutachter:

Prof. Dr.-Ing. Wolfgang Lehner
Technische Universität Dresden
Fakultät Informatik, Institut für Systemarchitektur
Lehrstuhl für Datenbanken
01062 Dresden

Prof. Dr. Carsten Binnig
TU Darmstadt
Fachbereich Informatik
Data Management Lab
64289 Darmstadt

Tag der Verteidigung: 26. Oktober 2018

Dresden im November 2018

ABSTRACT

The key objective of database systems is to *reliably* manage data, while *high query throughput* and *low query latency* are core requirements. To date, database research activities mostly concentrated on the second part. However, due to the constant shrinking of transistor feature sizes, integrated circuits become more and more *unreliable* and transient hardware errors in the form of multi-bit flips become more and more prominent. In a more recent study (2013), in a large high-performance cluster with around 8500 nodes, a failure rate of 40 FIT per DRAM device was measured. For their system, this means that *every 10 hours* there occurs a single- or multi-bit flip, which is unacceptably high for enterprise and HPC scenarios. Causes can be cosmic rays, heat, or electrical crosstalk, with the latter being exploited actively through the RowHammer attack. It was shown that memory cells are more prone to bit flips than logic gates and several surveys found multi-bit flip events in main memory modules of today's data centers. Due to the shift towards in-memory data management systems, where all business related data and query intermediate results are kept solely in fast main memory, such systems are in great danger to deliver corrupt results to their users. Hardware techniques can not be scaled to compensate the exponentially increasing error rates. In other domains, there is an increasing interest in software-based solutions to this problem, but these proposed methods come along with huge runtime and/or storage overheads. These are unacceptable for in-memory data management systems.

In this thesis, we investigate how to integrate *bit flip detection* mechanisms into in-memory data management systems. To achieve this goal, we first build an understanding of bit flip detection techniques and select two error codes, AN codes and XOR checksums, suitable to the requirements of in-memory data management systems. The most important requirement is *effectiveness* of the codes to detect bit flips. We meet this goal through AN codes, which exhibit better and *adaptable* error detection capabilities than those found in today's hardware. The second most important goal is *efficiency* in terms of coding latency. We meet this by introducing a fundamental performance improvements to AN codes, and by vectorizing both chosen codes' operations. We integrate bit flip detection mechanisms into the lowest storage layer and the query processing layer in such a way that the remaining data management system and the user can stay oblivious of any error detection. This includes both base columns and pointer-heavy index structures such as the ubiquitous B-Tree. Additionally, our approach allows *adaptable*, on-the-fly bit flip detection during query processing, with only very little impact on query latency. AN coding allows to *recode* intermediate results with virtually no performance penalty. We support our claims by providing exhaustive runtime and throughput measurements throughout the whole thesis and with an end-to-end evaluation using the Star Schema Benchmark. To the best of our knowledge, we are the first to present such holistic and fast bit flip detection in a large software infrastructure such as in-memory data management systems. Finally, most of the source code fragments used to obtain the results in this thesis are open source and freely available.

ACKNOWLEDGEMENTS

First of all, I thank my adviser Wolfgang Lehner for accepting me into the database group and giving me the opportunity to work on this thesis. At times it was not certain that I would successfully finish this work, but he believed in me and gave me support to finally submit the thesis.

Second, I am very grateful to Dirk Habich and Carsten Binnig for co-advising my thesis, as well as for all the valuable discussions and hints. Furthermore, I thank the whole database group for the great atmosphere, censorious opinions on talks and papers, as well as the fun with lab coats, Nerf guns, carrera race tracks, Mario cart tournaments and so on. In particular, I thank my colleagues Dirk (again), Schlegi, Kiki, Tobi, Martin, Kai, Claudio, Hannes, Ulrike (both), Jules, Johannes (both), Zippi, Patrick, Annett, Alex, Juliana, Lars (both), Robertron, Maik, Timbo, Katrin, and everybody else.

I am deeply thankful to my family, especially my wife Anja and son Dante, our parents and the whole family, for supporting me, lending me strength, keeping many things away from me so I could concentrate on my work, so very often. At times, it was difficult to unite my obligations to the family and the commitment to the PhD thesis and I cannot thank you enough for your support and your sacrifices.

Finally, I am grateful to all my friends who were responsible for the other, non-research part of my life and whom I can still call friends despite sometimes long times of not sending any signs of like ;-) Especially, I thank Tommek, Jurki, Benno, Gorath, Hannes, Robbi, Modro.

Till Kolditz
20th August 2018

CONTENTS

1

INTRODUCTION

The key objective of database systems is to *reliably* manage data [159], where *high query throughput* and *low query latency* are core requirements [2]. To satisfy these requirements, database systems constantly adapt to novel hardware features [22, 31, 42, 78, 107, 135]. In the recent past, we have seen numerous advances, in particular with respect to *memory*, *processing elements*, and *interconnects* [28, 67, 142]. These improvements effectively enabled the introduction of in-memory data management systems (DMSs), where all business and query-related data is completely kept in fast main memory. The respective hardware developments were enabled by the constant decrease of transistor feature sizes. Despite the hardware improvements, we have also known for a long time that hardware is *unreliable*. In the database community, research on hardware errors was restricted to a few classes [62]. However, the constant shrinking of transistors spawned a new type of errors, that of *transient soft errors* also known as *bit flips* [130, 198].

Although it was intensively studied and commonly accepted that hardware error rates increase dramatically with the decrease of the underlying chip structures [27, 68, 172], most database system research activities neglected this fact. Traditionally, the focus has been put on improving performance characteristics, in terms of increasing throughput or decreasing latency, by exploiting new data structures and algorithms. Meanwhile, the detection and correction of soft errors in the hardware was left to the hardware and operating systems communities. Especially for main memory, undetectable silent data corruption (SDC) may occur as a result of transient bit flips. Such faulty data is today mainly detected and corrected at the dynamic random access memory (DRAM) and memory-controller layer [172]. However, since future hardware becomes less and less reliable [26, 68, 150, 166] and since error detection as well as correction by hardware becomes more expensive [81], this free-ride will come to an end in the near future.

Borkar found that, statistically, the relative failure rate *per transistor increases* by 8 % per bit and technology node generation [26]. Since the density of both logic and memory circuits increases *exponentially*, soft error rates of the whole chips increase exponentially as well, as Figure 1.1 illustrates [26]. We give a recent example for the exponential increase in main memory size. In 2010 the first 8GiB DRAM modules were available and on a single central processing unit (CPU) socket with 8 DRAM slots, this allows 64GiB of total main memory. More recently, Samsung started mass production of their first 128GiB double data rate-4 (DDR4) DRAM modules in late 2015 [44]. For a single CPU socket with 8 DRAM slots, this allows 1TiB of main memory, which corresponds to an increase to 16× the capacity per module in about 5 years, or a 1.74× increase per year. In his paper, Borkar predicted only for technology nodes up to 16nm, while today vendors already ship products with transistor feature sizes of 10nm and below[146].

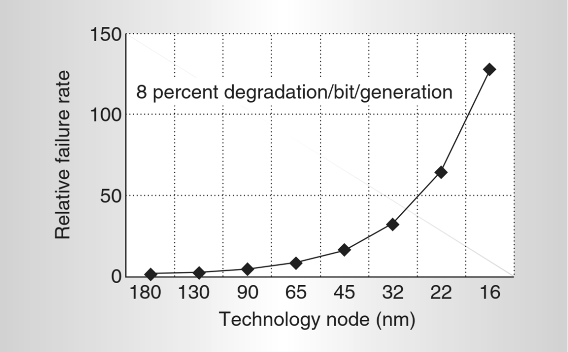


Figure 1.1: Soft-error failure-in-time of a chip, from [26]

Sridharan *et al.* conducted a field study on DRAM and static random access memory (SRAM) memory faults and errors in [173]. They collected data over several years from two large production systems, comprising in total more than 45 Billion DRAM device-hours. They find that faults in SRAM cells, typically used in CPUs, might not pose a significant threat to system reliability. In contrast, they conclude that hardware faults in DRAM will be much more significant and will require new error handling techniques. Even more alarming is the observation that the single error correction, double error detection (SECDED) capabilities of current error correcting code (ECC) main memory modules may be too weak for future error rates. Sridharan *et al.* measured *undetected* error rates of around 40 FIT[1] [174] per memory device (a single chip), which they describe as "*unacceptably high [...] for many enterprise data centers and high-performance computing systems*" [173]. Regarding the system where this error rate was measured, this translates into a single or multi-bit flip *every 10 hours*. More studies show findings which support the negative trend of the degrading memory reliability, as well as the fact that *multi-bit flips* make up an increasing, non-negligible part of all errors [72, 164, 165, 173]. Additionally, the wear and tear of circuits leads to *varying* error rates and increasing numbers of bit flips *during a system's life time*, where heat stimulates this effect [68]. Now, since main memory-centric data management systems have become the de-facto standard with all major vendors shipping some type of in-memory DMS [40, 47, 95, 147], the reliability of all the data kept in main memory is in greatest peril.

In this thesis we investigate the detection of transient hardware errors – bit flips – in in-memory data management systems, as a first step towards hardware error tolerant DMSs, where we concentrate on bit flips in data, i.e. data caches in the CPU, in main memory, and interconnects between these. To achieve this, we use two error control codes, which are applied to the managed data. An exceptional challenge is to keep the unavoidable overheads, in terms of additional memory consumption and processing time, as low as possible. Especially, in-memory DMSs almost exclusively work in fast main memory and avoid slow I/O to secondary storage like hard disk drives (HDDs) or solid state disks (SSDs). To keep the processing time overhead as low as possible, we investigate in detail at which places which of the error codes is beneficial. The memory overhead is directly linked with the detection capabilities and we will show the trade offs among which one has to choose for trading storage and runtime overheads against reliability.

1.1 CONTRIBUTIONS OF THIS THESIS

The contributions of this thesis can be summarized as follows.

(1) Lead to a New Database Research Domain for handling multi-bit flips which occur in several hardware components and which can not be handled by contemporary bit flip handling techniques. In the database domain there has not been the need to care about bit flips, because the bit flip rates and weights have been low enough so that hardware and operating system (OS) could mask such errors. With the prospect of increasing bit flip rates, it will be a necessity to deal with bit flips also on the software layer, to retain both *efficient* and *effective* error handling. In this course, we formulate five Requirements $\mathcal{R}1$ to $\mathcal{R}5$ for handling bit flips inside in-memory DMSs, and offer first solutions for the detection of multi-bit flips. With this work, we hope to pave the way for future researchers so that they build upon, extend, or complement the approaches presented in this thesis.

[1] One FIT is one error in one Billion device-hours.

(2) Detailed Analysis of Coding Techniques with regard to the data management domain for detecting bit flips. We investigate the general *applicability* of existing approaches, where we conclude that these are insufficient and that we need data management system-specific techniques based on data coding schemes. We select three schemes for which we conduct detailed *runtime measurements* for many operations used in database systems. An arithmetic error coding called AN coding, as well as bit-wise exclusive OR (XOR) checksums turn out to be suitable candidates for in-memory DMS-specific software-based bit flip detection.

(3) Silent Data Corruption Probabilities are devised and compared for the three selected coding schemes. In contrast to related work, we provide optimality criteria for selecting best AN coding parameters. Additionally, we distinguish between "super As" and "golden As" as the *set* of candidates for the best parameter and the best parameter under a specific optimality criterion, respectively. As compression plays a crucial role in modern DMSs, we provide golden As for all data bit widths between 1 and 32 bits and all parameter A bit widths between 2 and 16 bits, which constitutes unprecedented coverage. For that to achieve, we furthermore introduce a new approach for computing these probabilities for non-linear codes using *approximation on GPU-clusters*. We show how to transform our counted conditional probabilities into unconditional ones, exemplarily for the binary symmetric channel (BSC). By that, we provide a toolbox for choosing appropriate coding schemes based on desired data bit widths, redundancy margins, and error detection capabilities.

(4) AN Coding Improvements are developed which make it feasible to employ this software coding approach for in-memory DMSs. By that, AN coding allows error detection on value granularity at comparable speeds of block-based techniques like checksums. These improvements also allow pervasive *vectorization* of this code family, which is an important aspect for in-memory DMSs. We recapitulate vectorization of the selected approaches and refine it for AN coding since for error detection this is enabled only by our coding improvements. Compared to original AN coding, we improve throughputs for AVX2 execution to factors of up to $18\times$.

(5) Bit Flip Detecting Storage is introduced, based on AN coding and XOR checksums, where we focus on columns and pointer-based index structures. We will show that we only need to adapt the lowest layer of a DMS and that our techniques can seamlessly be combined with many of the modern data representation techniques. Due to the pervasive use of compression in modern DMSs, mostly only integers are stored in columns, and for that, our improved AN coding is a perfect fit for bit flip detection. As a consequence, all base columns and intermediate results can be completely AN encoded.

(6) Bit Flip Detecting Query Processing builds upon the bit flip detecting storage concept, making column-store query processing aware of error detection techniques. We identify several *opportunities* for error detection, where we show that a *continuous*, pervasive approach can be employed for *end-to-end* detection on each and every data element. Since AN coding allows to execute arithmetic operations directly on code words, a very tight integration of coding in physical query processing operators is possible. This enables continuous bit flip detection at run-time during query processing.

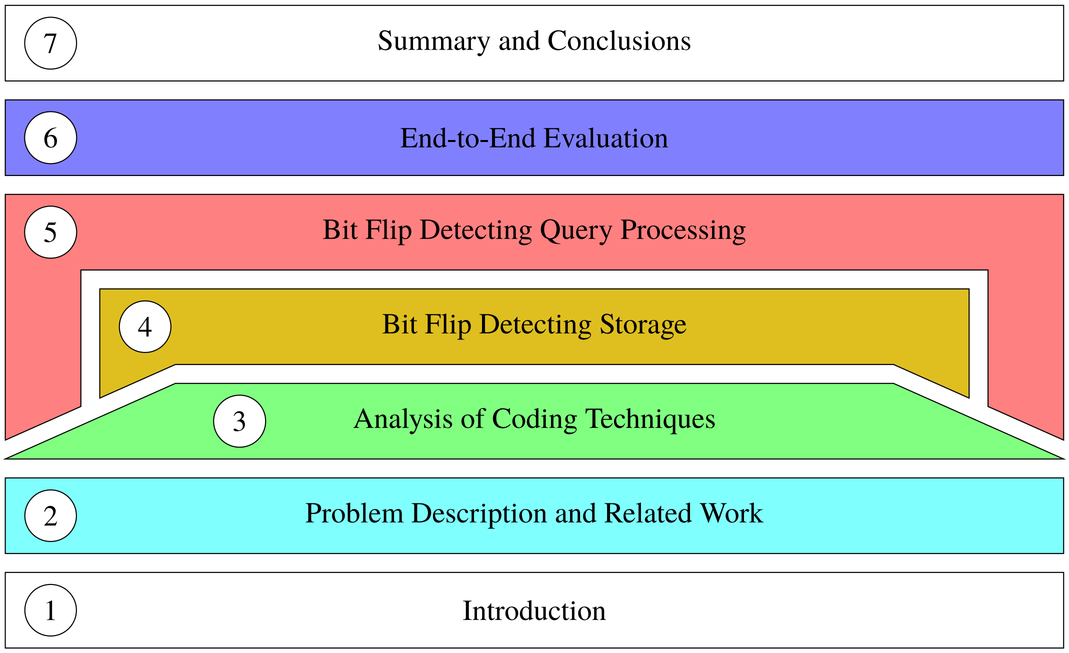

7
Summary and Conclusions
6
End-to-End Evaluation
5
Bit Flip Detecting Query Processing
4
Bit Flip Detecting Storage
3
Analysis of Coding Techniques
2
Problem Description and Related Work
1
Introduction

Figure 1.2: Structure of the thesis.

(7) Adaptable Error Detection is enabled through the use of AN coding and our findings for the silent data corruption probabilities. Its single code parameter A can be chosen depending on the data width and hardware error model, so that a desired minimal detectable bit flip weight is guaranteed by the code. Our coding improvements also allow during query processing to re-encode data on the fly with little to no runtime overhead. This aspect is very important, because it is highly expected that the error model changes during a DMS's life-time and bit flip detection techniques must therefore be adaptable at run-time, too.

(8) End-to-End Evaluation shows that our developed concepts greatly reduce the work and storage overheads compared to existing software-based, general-purpose approaches. Especially, double or triple modular redundancy introduce more than 2- and 3-fold overheads, respectively, regarding all aspects of storage and query execution. In contrast, our approach shows execution overheads of only 20% on average for the star schema benchmark (SSB) and storage overheads adjustable to the actual error model for error detection.

1.2 OUTLINE

The structure of this thesis is illustrated in Figure 1.2 and in great part reflects the contributions. First, in Chapter 2 we discuss the problem of multi-bit flips, then make the reader acquainted with the topic of error handling in general, and introduce known techniques from the hardware stack. We will then formulate a set of requirements for software-based approaches and discuss existing ones. We will see that none of them meets our special requirements. The major outcome is, that replication-based techniques are not suitable in our context and that the only alternative is *error control coding*. Based on this, in Chapter 3 we dissect three software-based data coding techniques in greater detail, which includes discussion on coding operations and their performance in terms of *silent data corruption*, memory consumption, and coding runtimes. We afterwards present new improvements for AN coding, which allows vectorization of all code operations. As the major outcome, we will regard AN coding and XOR checksums for later use. Afterwards, we discuss our hardened storage concepts in Chapter 4, where we integrate the two error control codes in the storage layer. We will show that, the codes should be used at different places. Building upon that, we introduce in Chapter 5 our hardened query processing concepts, where we introduce three *opportunities* at which error detection can take place, where all of them show different advantages and disadvantages. To proof the feasibility, we conduct an end-to-end evaluation in Chapter 6, where we first discuss our prototype implementation and then present runtime measurements first for individual query operators and then all of the SSB queries in total. Finally, we conclude the thesis in Chapter 7 where we also provide directions for future work.

2

PROBLEM DESCRIPTION AND RELATED WORK

In this chapter, we provide a detailed problem statement and discuss why transient hardware errors in the form of bit flips are an important challenge. Therefore, in Section 2.1 we first revisit the reliability assumptions and considered failure classes in DMSs and how research in the database community concentrated only on the efficiency aspect of data management. Then, we elaborate on how hardware becomes less and less reliable in Section 2.2, where we also show several causes which were confirmed by past research. This leads to an increase in transient error rates, which requires error handling techniques. We will shortly present a taxonomy on such techniques to clarify and introduce several terms. Afterwards, we will present approaches trying to mitigate bit flips in hardware Section 2.3. As we will see, these hardware-based techniques will not scale on their own to accommodate the increasing error rates. We will then derive requirements for a software-based solution in Section 2.4. Based on these, we evaluate research from software communities for dealing with the increase in transient errors in Section 2.5. As we will see, these are also inappropriate for the data management domain and we will conclude that we need specific techniques for tackling transient hardware errors in main memory-centric data management system. Section 2.6 summarizes and concludes this chapter.

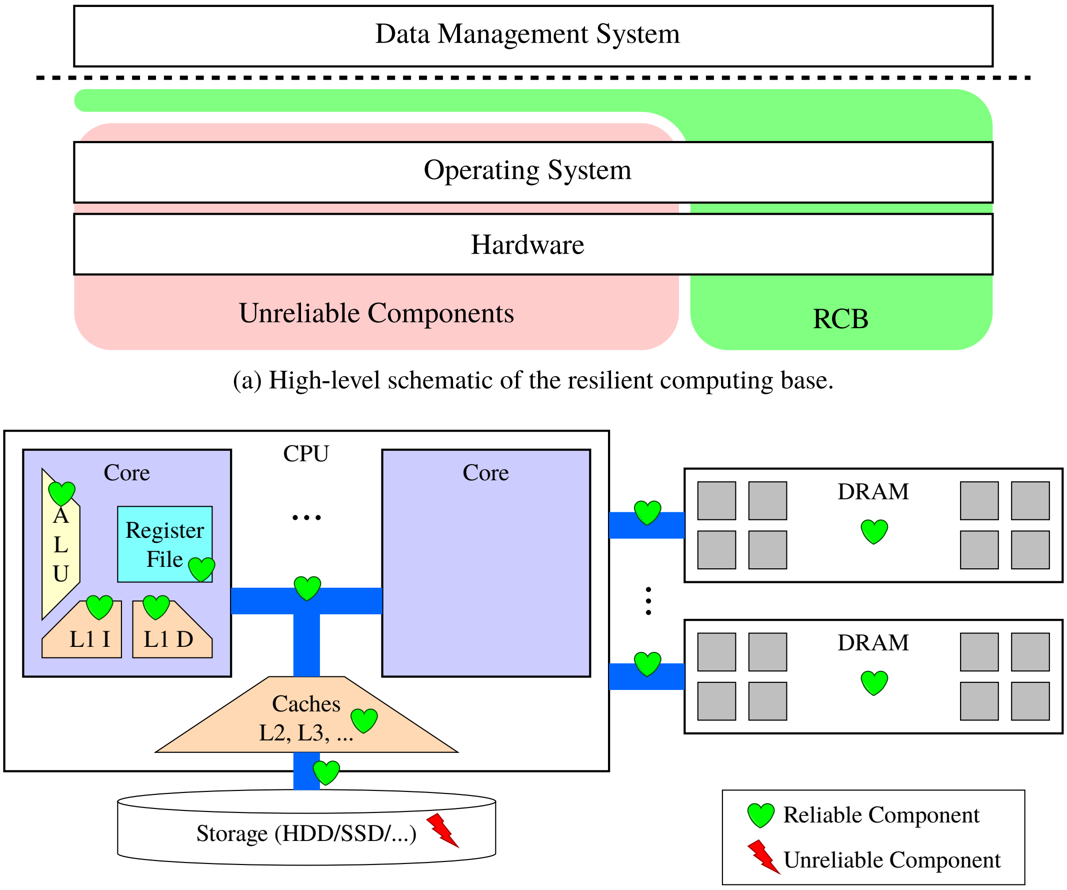
2.1 RELIABLE DATA MANAGEMENT ON RELIABLE HARDWARE

The most important objective of data management is the *reliable* data storage and query processing [159]. This is a nontrivial task, because all hardware and software contain flaws. For instance, on a HDD data tracks may become corrupt and therefore HDDs employ ECCs to recover those tracks, which is the same for storage medias like CDs, DVDs, and the like. Still, even HDDs can have undetected errors in their data, which requires additional software techniques to detect them [170]. Also, more modern storage devices like SSDs require strong error correction codes to guarantee correct retrieval of data, e.g. recent ones correct up to 550 bit errors for each 4KB block of data [8][1]. The OS works as an abstraction layer between hardware and applications and does not only provide generic interfaces to hardware, but also tries to mask as many remaining errors as possible. For instance, device drivers may have errors due to software bugs or erroneous hardware states and the operating system can, instead of stopping user processes, restart that device driver without the application ever noticing it [38]. An OS, in turn, requires some minimal amount of reliable hardware components, which allow it to function properly and which allow masking of certain errors. For this minimal set of components, in the operating systems domain, Engel and Döbel introduced the term *Reliable Computing Base* [46], which they define as follows.

Definition 1 (Reliable Computing Base). *"The Reliable Computing Base (RCB) is a subset of software and hardware components that ensures the operation of software-based fault-tolerance methods and that we distinguish from a much larger amount of components that can be affected by faults without affecting the program's desired results." [46]*

Figure 2.1a illustrates how we understand the RCB. On the one hand, some hardware and OS components may be unreliable in the sense that they can not or should not handle all possible errors themselves. On the other hand, there are hardware and OS components which handle and mask the errors to provide a reliability abstraction (denoted as dashed line) to applications and data management systems, in particular. One important type of errors are *transient bit flips*, which were observed already several decades ago. There, e.g., the main memory state is changed on the physical level due to environmental influences and not due to software-issued write operations [27,

[1] There are no numbers given how much additional, redundant code bits are required to achieve this correction rate.



(a) High-level schematic of the resilient computing base.

(b) Detailed view of hardware components assumed to be reliable or unreliable.

Figure 2.1: The contemporary RCB (a) and more detailed view of the hardware components (b). In essence, all components except the mass storage devices are assumed reliable.

130, 150, 165, 197]. The danger of bit flips is that they change data in a *silent* way, so that if not detected, applications work on wrong data. Solutions were developed at the hardware and OS level. On the hardware level, SECDED ECC DRAM was introduced [198], where today for any 8-byte block all single bit flips can be corrected and all double bit flips can be detected. This may happen either in the DRAM modules or in the memory controller, which nowadays typically resides inside the CPU. When, however, a non-correctable bit flip is detected, the operating system receives a hardware signal and typically terminates the affected process. The OS can detect bit flips at some occasions, e.g. when a process's virtual memory mapping table becomes corrupt. Again, when errors are detected there, the OS typically terminates the affected process. More recently, there were proposed automatic OS-based replicated execution schemes for arbitrary applications to mitigate effects of bit flips [43]. As a consequence, in the database community the problem of transient bit flips was left aside and was paid little to no attention during the last three decades. Figure 2.1b shows in more detail some hardware components which are of interest in the case of main memory-centric DMSs, as well as whether they are today assumed reliable from the DMS perspective. The CPU and its sub-components, as well as main memory, which is typically DRAM, and the interconnects between them are assumed reliable in the sense that the application (DMS) is shielded from transient errors, as described before. However, as we will see in an instant, storage devices are assumed unreliable in the sense that single tracks or sectors may become corrupt over time, or that a whole device may fail.

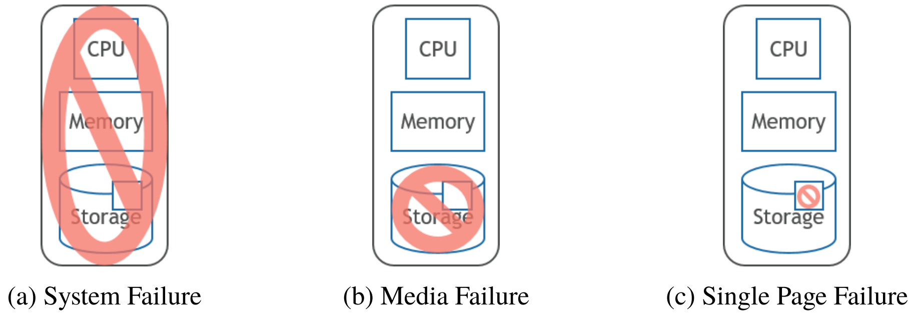
The RCB can also be seen respectively from the view of applications, where (part of) the OS is included as software component in the RCB. In the remainder of this thesis, we will use this broader sense of an RCB shown in Figure 2.1a, as it better depicts the view from a data management system. DMSs implicitly assume an RCB, but the fact that not all errors can be masked by hardware and OS poses a serious threat to the reliability assumption of DMSs. By that, all along, data management systems had to deal with various types of errors or *failures*. There have mainly been considered four *classes* of failure so far [62], with three of them visualized in Figure 2.2. These first three are concerned with physical (hardware) failures and are most important for our case.

(a) **System Failures** occur due to a power failure (loss of power), where the volatile whole memory state becomes lost. As a consequence, DMSs keep a persisted database on some non-volatile storage media, e.g. HDDs or SSDs.

(b) Hardware may fail due to manufacturing failures or wear and tear and this especially affects the mass storage devices which hold the persisted database state. This is reflected in Figure 2.1b, where all hardware components except the mass storage devices are assumed reliable. Consequently, part of or the whole non-volatile storage may be lost, which is called **Media Failure**. Solutions are, e.g., to use Redundant Array of Independent Disks (RAID) systems [136] or additional archive and log backups.

(c) Besides power and media failures, there may be errors during writing the persistent state onto storage devices. For instance, an error in a storage device's firmware or in the OS may lead to *partial* writes, which leads to inconsistent states. This may also happen due to a power loss, where not all desired content could be written to disk in time. These problems are called **Single-Page Failures**. One proposed solution for this class is to use page recovery indexes and rebuild pages from older versions and the database log.

The fourth and last class are *transaction failures*, which occur due to multiple concurrent users accessing the same data. This class does not affect the reliability requirement of data management with regard to hardware errors, which is why we do not further consider this last class. All these failure classes have in common that DMSs have made the *implicit assumption* that the underlying systems guarantee a certain degree of reliability – the RCB. For instance, when there is a power failure and a system is restarted, then *all* volatile memory is lost, and not just a part of it. A DMS can then detect that it restarted and apply appropriate measures to fix any data inconsistencies. Likewise, errors could be reliably attributed to one of the failure classes. This is also known as fail-stop behavior. Consequently, as Figure 2.1b showed, DMSs today assume CPU, main memory, and interconnects reliable, whereas storage devices are considered unreliable.



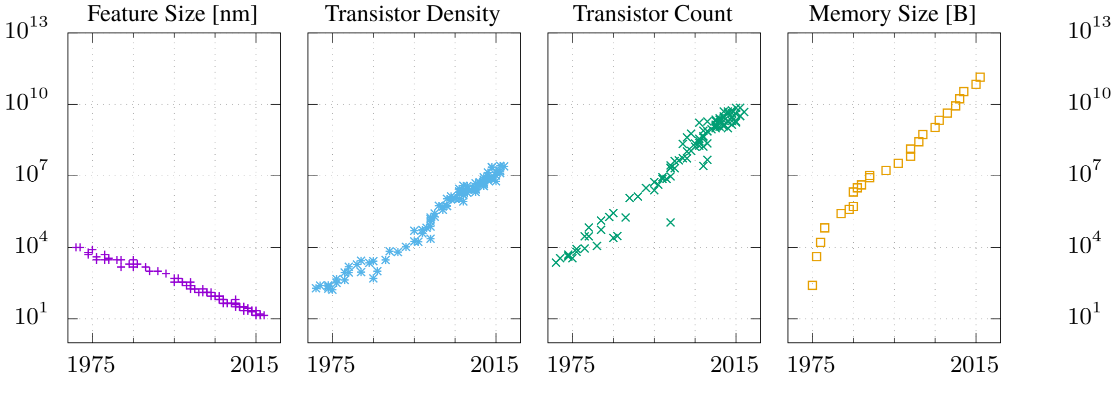
(a) System Failure (b) Media Failure (c) Single Page Failure

Figure 2.2: Contemporary data management failure classes (a), (b), and (c), after [62, Figure 1].

By taking care of the above failure classes, contemporary DMSs could meet the reliability requirement. By that, the community could mainly focus on the second most important requirement of data management: the *efficient* data storage and query processing [2]. First of all, maximum query throughput, maximal query concurrency, and minimal query latency are desired. Second, the memory footprint should be kept as low as possible, to keep as much data in fast main memory as possible and to allow as many parallel transactions as possible, since each query requires some private memory. Third and finally, DMSs are constantly adapted to novel hardware features [22, 31, 42, 78, 107, 135], which in turn typically enable to improve throughput, latency, concurrency, and memory footprint. In the recent past, we have seen numerous advances, in particular with respect to *memory*, *processing elements*, and *interconnects* [28, 67, 142]. For instance, vector instruction sets allow to process many data elements in parallel by single instructions. In the previous decades, these hardware advancements also have led to a shift in the community towards *in-memory database systems* [21, 100, 101, 176]. As main memory has become ever cheaper and denser, this allowed to store and process most if not all business data completely in main memory. By that, the efficiency of data management could be greatly improved, and e.g. new main memory-centric data structures [39, 83, 102] and data storage formats [6, 36] were developed.

2.2 THE SHIFT TOWARDS UNRELIABLE HARDWARE

Basically, the aforementioned hardware speed improvements and increasing amount of features supported by novel hardware, are possible due to the constant shrinking of transistors. Since several years already, we speak of *nano-scale* transistors and the *nano-era*, where, today, integrated circuit manufacturers already produce transistors with feature sizes of 10nm and below [146]. For the last decades, about every 18 months the transistor feature size could be decreased so far that the transistor density could be doubled, also known as “Moore’s Law” [121] (cf. Figure 2.3a). As a consequence, the *transistor density* (cf. Figure 2.3b) and transistor count increased *exponentially* for both CPUs (cf. Figure 2.3c) and DRAM modules (cf. Figure 2.3d). By that, in turn, there has been a constant increase in computational power in terms of increasing frequencies, numbers of cores and hardware threads, ever extended instruction set architectures (ISAs) and vectorization capabilities. In particular, the hardware components CPU, main memory, and interconnects are of main interest here, being the main drivers for performance improvements.



(a) (b) (c) (d)

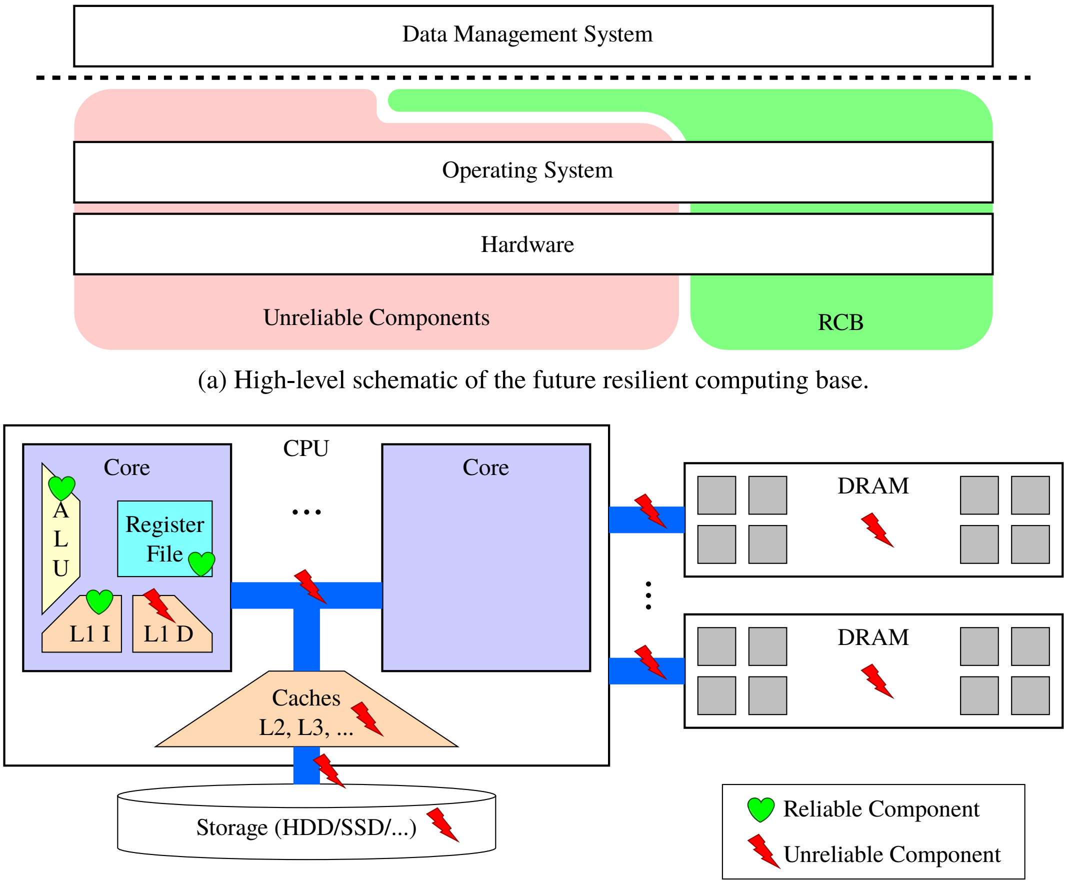
Figure 2.3: Historical development of (a) transistor feature size (technology node), (c) number of transistors (CPUs), (b) transistor density (CPUs), and (d) main memory module capacity.

Unfortunately, this leads to an *exponential* increase in transient bit flips [27, 68, 172]. The transistor feature size reduction leads to less and less atoms storing the electrical charge required for distinguishing between the off and on states. As a result, ever smaller voltage ranges drive the toggling of the transistors and, by that, external influences more easily affect a transistor's state [150]. This is anything but new as such effects have been observed already since the late seventies [11, 30, 130, 197]. Hardware-based protection like parity bits and ECCs helped to retain the RCB, but scaling up such techniques is very challenging [68, 150, 166]. The following *external influences* are suspected to be the main causes for bit flips [82, 150]:

1. cosmic rays (or energetic particles in general),
2. manufacturing defects (e.g. transistor variability),
3. heat and heat gradients, as well as
4. disturbance errors (electrical cross-talk).

When hit by a cosmic ray, for instance, a charge could be induced in a so far non-conducting transistor or a memory cell, or it could cause a short circuit leading to discharge of a DRAM cell's capacitor. Heat directly influences the conductivity and threshold voltage of transistors [163]. Electrical cross-talk means, e.g., that reading or writing one memory cell can change the state of surrounding memory cells [125]. Zhu *et al.* showed that circuits also become more susceptible due to energy-saving techniques such as voltage and frequency scaling [196]. Finally, transistor variability may result in largely varying gate switch delays [127], or lead to varying sensitivity to the aforementioned causes of error. While the rate rises how often bit flips appear, *multi-bit flips* also become more and more prominent and studies showed that they already make up a non-negligible amount of total bit flips [82]. Kim *et al.* found that all newer DRAM modules are affected and they observed one to four bit flips per 64 bit word even for ECC DRAM [82]. This is more than such DRAM modules can reliably detect. Several studies proved that hardware components become more susceptible in total to bit flips, which is already observable today in large clusters and computer farms [72, 164, 165]. A contemporary assumption is still, that at most a single bit flips per process execution. This is an error model called *single event upset* and is to become obsolete, as the above shows. On top of that, the error behavior *changes at run-time*, especially due to hardware aging effects [150], where heat further stimulates this [68]. In summary, bit flip rates increase exponentially and multi-bit flips become more and more frequent, while aging effects worsen the situation and lead to error models varying at system run-time.

While all hardware components are affected, *memory cells* are more susceptible than logic gates [68, 72, 81]. The former are much more vulnerable to electromagnetic interference effects [82, 125] as well as to data retention problems, because they have to store their state for an arbitrarily long duration. A practical solution to tackle interference effects in DRAM is to increase the DRAM refresh rate, so that the probability of inducing disturbance errors before DRAM cells get refreshed is reduced [82, 125]. However, refresh operations waste energy and degrade system performance by interfering with memory accesses [82, 125]. In addition, with increasing DRAM device capacities, the following effects with regard to DRAM refresh arise at the same time as well. (*data retention problem*) [79, 80, 110, 125]. First, more refresh operations are necessary to maintain data correctly. Second, smaller DRAM capacitors require lower retention times [125]. Third, the voltage margins separating data values become smaller, and as a result the same amount of charge loss is more likely to cause multi-bit flips [79, 80, 110, 125]. Theses effects increase as DRAM device capacities increase [79, 80, 110, 125] and in a recent field study, more than 20% of the observed errors were large multi-bit faults [173].



(a) High-level schematic of the future resilient computing base.

(b) Detailed view of hardware components assumed to be reliable or unreliable.

Figure 2.4: The future RCB (a) and the reliability of its components (b).

Furthermore, emerging non-volatile memory technologies like PCM (phase change memory) [98], STT-MRAM [91], and PRAM [191] exhibit similar and perhaps even more reliability issues [79, 80, 110, 125]. For example, PCM are based on multi-level cells (MLC) to store multiple bits in one cell which is achieved by using intermediate resistive states for storing information, in addition to the low and high resistance levels [118]. A major problem in PCM is that a time-dependent resistance drift can have the effect that a different value than the originally stored one will be read [118]. Furthermore, if one cell drifts to the incorrect state, other cells are also highly likely to drift in the near future. Due to this correlation, multi-bit flips at run-time are much more likely in MLC PCM [118]. Moreover, heat produced by writing one PCM cell can alter the value stored in many nearby cells [118]. This was observed for up to 11 cells in a 64 byte block [77].

As a major consequence, the RCB from Figure 2.1 is greatly affected and changes. There will be a growing proportion of the errors which the RCB cannot mask any longer, as depicted in Figure 2.4a. The increasing number of unreliable components is detailed in Figure 2.4b: While today DMSs assume errors only in mass storage devices (cf. Figures 2.1b and 2.2), in the future they have to assume errors in central components like the CPU, main memory, and interconnects. Regarding the CPU, for now we assume that only the data cache can be affected. We will consider later what happens if the CPU becomes even less reliable and how we can detect errors in arithmetic operations.

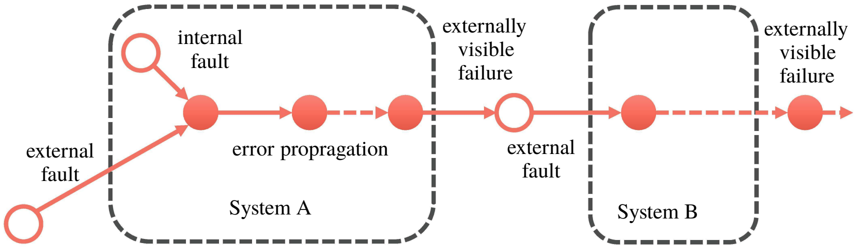
To deal with these issues, some form of *error handling* is required. Error handling in general is a very broad field, for which Avižienis *et al.* compiled a taxonomy of error types and countermeasures in general [12]. We will now shortly make the reader familiar with several terms used throughout this thesis and relate them to this work. First of all, there is an important distinction between faults, errors, and failures (cf. Figure 2.5a). In short, from a (sub-)system's view, there may be some external or internal *fault*, which may lead to an internal erroneous state – the *error*. This error, in turn, may lead to a *failure* of that (sub-)system, where an abnormal behavior is observed by other (sub-)systems. This failure may then be the external fault for some other (sub-)system so that the initial fault propagates through several layers of a larger system. For an example, consider the case for memory cells as depicted in Figure 2.5b: the initial fault may be a cosmic ray (external) or a damaged flip-flop transistor (internal). This in turn may trigger the internal state to change from $0 \rightarrow 1$ or $1 \rightarrow 0$ and, actually, a bit flip occurs – an error in the memory cell. When this cell's state is read, this internal error manifests as a *failure* to the environment, which in turn may become an external fault from the perspective of an application. For instance, when this memory cell held part of a pointer stored by the application, the pointer may now contain an invalid address not yet allocated to the application's virtual memory space. Trying to read or write at this invalid address triggers an actual error in the application, because the operating system will now emit a `SIGSEGV` segmentation fault signal. This leads to a failure which, finally, the user sees as an error message and which typically results in the operating system forcing termination of the affected application. This example shows the significance of the problem of bit flips: they may activate errors on many different system layers and they may stay undetected for an arbitrary amount of time. Furthermore, Avižienis *et al.* distinguish between *transient* (soft) and *permanent* (hard) errors (cf. also [20]), where e.g. the latter can be a permanently faulty memory cell which cannot store any charge (in the case of DRAM). Based on this, we regard bit flips as the following:

Definition 2 (Bit Flip). *A bit flip may be any of a fault, error, or failure, depending on the (sub-) system we look at and, consequently, we will use it as synonym for any of the three terms. In this thesis, we are exclusively considering* physical *faults, which includes some types of faults which are also categorized as* development *faults and* interaction *faults [12]. Examples for physical faults are: production defects, physical deterioration (e.g. aging), or physical interference (e.g. electrical cross talk such as RowHammer [125]). We do not consider faults such as software flaws or bugs, logic bombs, hardware errata, intrusion attempts, viruses or worms, or input mistakes [12, Figure 5 (a)]. In this thesis we do not explicitly distinguish between transient and permanent faults.*

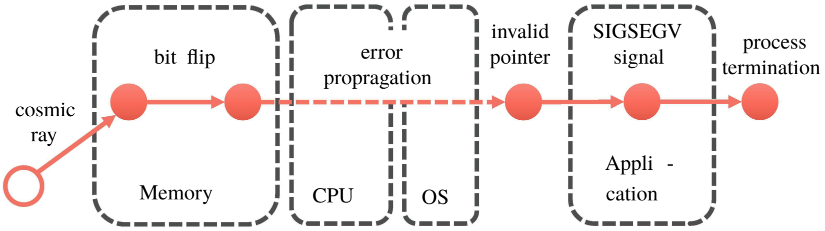
Whether distinguishing between transient and permanent faults is necessary, whether it can really help in our context, and, if so, how to do the distinction is deliberately subject to future work. Throughout the remainder of this thesis, we will always have to relate to the *number* of bits that flip in a data unit, which we define as follows:

Definition 3 (Bit Flip Weight). *We denote the number of flipped bits per data unit, e.g. a data or code word, as bit flip weight (BFW). As symbol we use b, which means that b bits flipped in that data unit.*

Dependability describes which guarantees a system delivers with reference to some error model. An error model, in turn, characterizes the anticipated errors that may occur. Dependability is only a very abstract term and Avižienis *et al.* consider 6 attributes in their taxonomy [12]: availability, reliability, safety, confidentiality, integrity, and maintainability. While an error class such as hardware errors may concern several of these classes, especially safety, in this thesis we concentrate on the *reliability* attribute. To achieve reliability, Avižienis *et al.* classify available means into four categories: 1. fault prevention, 2. fault removal, 3. fault forecasting, and 4. fault tolerance. First, fault prevention may occur on the hardware level by eliminating the chance of transient errors, which however is not possible considering the current trends of hardware susceptibility. Second,



(a) Concept



(b) Example

Figure 2.5: The distinction between faults, errors, an failures and how it depends on the point of view.

fault removal means replacing faulty components, which implies down times when replacing integral pieces such as the CPU or main memory modules. Third, fault forecasting techniques try to estimate future incidences an likely consequences of errors to allow preventive actions such as component replacement. Fourth and finally, fault tolerance incorporates techniques which provide the ability to still deliver service even in the presence of bit flips. Especially in the data management domain, down times should be avoided so that fault tolerance techniques are most desirable. To achieve fault tolerance, Avižienis *et al.* further divide means into *error detection* and *recovery*. Naturally, we first have to detect that there actually is a bit flip. This may happen preemptively, where service delivery is suspended, or *concurrently* to the running service. Again, service suspension should be avoided and a concurrent detection technique is desirable. In this thesis, we will deal only with the first aspect, error detection. Furthermore, we strive to achieve *concurrent* error detection, because we believe that error rates and bit flip weights will otherwise increase too much to provide both efficient *and* reliable data management.

2.3 HARDWARE-BASED MITIGATION OF BIT FLIPS

Fault-tolerance techniques for countering the increasing numbers of (multi-) bit flips were proposed for the hardware layer. Regarding bit flips in memory, this mostly includes increasing the strengths of employed error correction codes [81]. This, however, is achieved at the expense of larger code blocks for which more bit flips are detected. For instance, Kim *et al.* investigate the use of 2-dimensional codes for In total, little to no more bit flips can be detected for the total amount of memory, but larger burst errors are then detectable with higher probability. The main problem here is the use of general purpose codes and worst case assumptions, because at the hardware level there is no knowledge about the actually running applications. Increasing the bit flip detection capabilities of these general purpose techniques is challenging and very expensive to cope with future transient error rates. The main bottlenecks are greatly increasing coding latencies and energy consumption [81]. Naseer and Draper suggest a double error correction (DEC) code for SRAM memory cells [126]. Compared to an SECDED implementation, their encoder requires between 70% and 107% more chip area and incurs up to 25% coding latency. Their decoder, in turn, adds between 359% and 12905% chip area and between 56% and 67% coding latency [126]. Apart from the fact that their approach is targeted for SRAM instead of DRAM, they buy their improvements in coding latency by a vast amount of chip area, where they store precomputed mappings for correction. That these precomputed mappings could themselves get corrupt over time is not part of their considerations.

Several other approaches were introduced to handle bit flips in the CPU. Mahmood and McCluskey use a *watchdog* processor for detecting errors [112]. This is a co-processor which monitors the system behavior and thereby performs error detection. The expected behavior must be provided to the watchdog co-processor beforehand, so that it can compare the desired and actual system behavior. This information may contain memory access behavior, control flow characteristics, expected control signals, or the reasonableness of results [112]. Since data management systems execute arbitrary user queries and store and process arbitrary data, it seems impossible to describe the required information about such a system for a watchdog processor. Austin introduced DIVA, which assumes a main high-performance, out-of-order core and a second, simpler core, which is used to validate the first core's execution [10]. Furthermore, they use a watchdog timer to detect deadlocks and livelocks in the CPU. However, this only deals with faulty execution, but not with faults in memory. Several real system implementations replicate part of or whole hardware blocks and use additional checkers to validate all the redundant computing. Examples are Compaq NonStop Himalaya [70], IBM S/390 [171], and the Boeing 777 airplanes [192, 193]. There are many other proposals which reduce overheads – in terms of number of hardware blocks, system complexity, monetary costs – by using the already available redundancy of modern multi-threaded or multi-core architectures – simultaneous multi-threading (SMT) – i.e. execution units in terms of redundant cores or hardware threads. Saxena and McCluskey were the first to exploit redundant threads for mitigating transient errors [160]. Rotenberg expanded SMT to AR-SMT by leveraging micro architectural improvements [157]. Reinhardt and Mukherjee introduced the concept of simultaneous redundant multi-threading (RMT), which increases the performance of AR-SMT by comparing redundant data streams before storing them to memory [151]. Vijaykumar *et al.* developed the SRTR processes, which again expands the RMT concept. They added fault recovery by delaying commits and possibly rewinding to a known good state [179]. All of these techniques introduce massive work overhead, as they all execute everything at least twice. This always reserves twice as many resources as are originally needed, even in the case of error-free execution. Even if the runtime overhead could be reduced by some of these, the total amount of work is increased several times. Consequently, such techniques incur overheads that are very high, either in terms of memory consumption or additional amount of work.

Meanwhile, other communities have well embraced the fact that hardware becomes unreliable [68, 150]. Since recently, a growing number of researchers strive to handle bit flips *also* at the upper layers of the hardware / software stack [150, 166]. One advantage is, that then, knowledge about the individual applications can be used to select the most *effective* and *efficient* solutions for handling (multi-) bit flips. For instance, hardware and operating system can not know about the actual type of data an applications stores or processes. These lower layers can only apply general purpose techniques, which are typically operating on fixed-sized blocks of data (e.g. ECC DRAM). In contrast, an application knows exactly what kinds of data and data structures it stores and processes and can therefore select bit flip handling techniques which can be more efficient or more effective than the general purpose techniques from the lower hardware/software layers. However, only hardware and OS have access to some data structures hidden from the applications, like virtual memory mapping tables. An application cannot handle errors there, especially because only the OS may in turn know about the exact implementation of such data structures. Another example is compiled, executable code: Application developers should not necessarily deal with protecting the executable machine code, but the compiler or OS should deal with this. Consequently, we share the belief that *all* the individual layers in the hardware / software stack must investigate for what part of a system they are actually responsible for, and in how far there are layer-specific techniques to deal with increasingly unreliable hardware.

2.4 DATA MANAGEMENT SYSTEM REQUIREMENTS

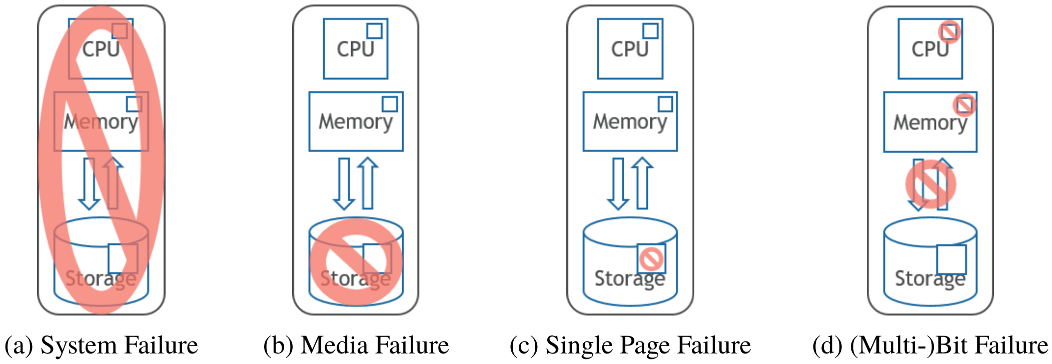
Scaling hardware-based solutions to tackle the increasing problem of transient bit flips is very challenging [68, 81, 150, 166]. The research of software-based solutions which aid the hardware and each other [68, 150] is a required step towards reliable data management. Contemporary in-memory DMSs are not prepared to handle these more and more frequently occurring, arbitrary bit flips in an effective or efficient manner. Generally, any undetected bit flip destroys the reliability objective of data management in the form of (i) false negatives (missing tuples), (ii) false positives (tuples with invalid predicates), or (iii) inaccurate aggregates in a *silent* way. As we showed above, replication techniques introduce very high overheads, either in terms of additional space or additional work. In order to cope with the increasingly unreliable hardware, we need to integrate bit flip detection into the DMS itself. Based on our observations, we define the following Requirements $\mathcal{R}1$ to $\mathcal{R}5$ which must be met by a software solution for tackling bit flips:

Requirement $\mathcal{R}1$ – Effectiveness
The reliability assumption of data management *must* be preserved under the assumption of increasing bit flip rates and weights. Since the RCB cannot mask all such errors, DMSs must introduce bit flip *detection* and *correction* mechanisms, which can handle given bit flip rates or weights. These mechanisms must be *effective* in the sense that they provide sufficient guarantees to detect (or correct) certain bit flip rates and weights.

Requirement $\mathcal{R}2$ – Efficiency
Bit flip handling techniques introduced in data management – a *software* layer – lead to an overhead in terms of query latency and memory footprint, which is *unavoidable*. This overhead must be as little as possible and must pay off hardware advances to be feasible. This means, bit flip handling techniques should introduce less overhead than future hardware generations bring speed and memory improvements. Since modern data management systems leverage hardware features such as vector instruction sets, error control techniques must be able to leverage such hardware features, too.

Requirement $\mathcal{R}3$ – Adaptability
Bit flip rates and weights will vary with hardware generations and during a system's life time due to hardware aging effects [150]. Therefore, bit flip handling techniques must be *adaptable* to changing error models *at run-time*.

Requirement $\mathcal{R}4$ – Availability
Especially in the data management domain, down times of systems should be avoided as much as possible, which for businesses otherwise means reduced income and increased operational expenses due to recovery times. Therefore, DMSs should be fault-tolerant and bit flip handling mechanisms should run concurrently to the normal operation – both in the sense of the taxonomy by Avižienis *et al.* [12] – to provide zero-downtime service.



(a) System Failure (b) Media Failure (c) Single Page Failure (d) (Multi-)Bit Failure

Figure 2.6: Extended data management failure classes (a), (b), and (c), including the new failure class of bit flips (d)

Requirement $\mathcal{R}5$ – Separation of Concerns
We support the idea of interdisciplinarity [150, 166], because there are aspects of bit flip handling which are out of the scope of our domain. Since we view the problem of bit flips from a data management-centric view, we should concentrate on protecting the business data and all data structures and temporary data which are used for query processing on the managed data. We can also safely assume that the actual data, both databases and query intermediate data, are much larger than the executable machine code. Furthermore, designing fault-tolerant algorithms – in addition to data structures – could be part of our domain, but are not considered in this thesis. Aspects like the control-flow of programs or data structures from other layers like the OS are out of our reach and scope.

Requirement $\mathcal{R}1$, *effectiveness*, is by far the most important one in this thesis. A DMS must be able to guarantee the detection of bit flips in its data to some desired degree. Otherwise, it would be superfluous. Next to the requirements, we also need an error model, which is abstract in the sense that, as we argued above, the actual bit flip weights and rates will vary during run-time.

Definition 4 (Thesis Error Model). *In accordance to Requirement $\mathcal{R}5$ and since memory cells are more susceptible to bit flips than logic gates, in this thesis we restrict the set of affected hardware components to (cf. Figure 2.4b):*

1. *CPUs, in particular the data caches,*
2. *main memory, mainly in the form of DRAM, and*
3. *interconnects between CPU and main memory.*

This means, we primarily care about error detection in our most precious good – data. It is furthermore not our task to detect bit flips in the executable code of a DMS, because this problem should be solved, e.g., by the compiler. Furthermore, we do not distinguish between transient and permanent bit flips, in accordance to Definition 2. We do not restrict the bit flip weight in our error model, because these numbers will depend on the actual hardware. Since solutions must be adaptable anyways (Requirement $\mathcal{R}3$), it would at most be reasonable to provide an upper boundary of the weights and rates, but that could even be different for the various hardware components.

The problem of bit flips has an impact on the previously presented failure classes in DMSs (cf. Figure 2.2), which we extend as shown in Figures 2.6a to 2.6c. On the on hand, while only CPU, (main) memory, and storage have been previously considered, we now also include the *interconnects*, depicted as double arrows. On the other hand, we consider bit flips and transient errors in general as a new *failure class* shown in Figure 2.6d. This failure class expresses our thesis error model described in Definition 4. Also, we will only consider a single server system and *no* distributed computer environments.

The goal of this thesis is to find or develop bit flip detection techniques suitable for data management, which satisfy all of the above Requirements $\mathcal{R}1$ to $\mathcal{R}5$ and which cover our (very generic) error model. In the following we will provide an overview of software-based techniques and show that, so far, there is none which satisfies all of our requirements. This, in turn, underlines the necessity of this research direction in the data management domain.

2.5 SOFTWARE-BASED TECHNIQUES FOR HANDLING BIT FLIPS

There are several software-based techniques which are directly targeted at or related to the handling of bit flips. In the following, we will have a closer look at operating system (Section 2.5.1), compiler level (Section 2.5.2), and application level techniques (Section 2.5.3) and we will evaluate them against our requirements Requirements $\mathcal{R}1$ to $\mathcal{R}5$.

One fundamental technique shared by virtually all of the following bit flip handling means is *n-way modular redundancy*, a generalization of triple modular redundancy (TMR) introduced by Lyons and Vanderkulk [111]. There, either hardware resources (resource redundancy), data storage (space redundancy), data processing (time redundancy), or any combination of these are *replicated* n times. Replication of data means to store the same data n times, while replication of hardware resources implies e.g. running n machines, or using n hardware threads to solve the same task. Additionally, the n must be compared to detect errors or to vote a majority to even correct errors. Hardware replication typically, but not necessarily, involves replication of data. Redundant processing can be done alone by using the same hardware to execute a task n times on the same data, but since this increases the runtime by a factor of n, typically, redundant hardware resources are used to not increase the runtime as much. Many applications of double modular redundancy (DMR) or TMR were proposed [43, 93, 94, 138, 170] and all show that the overhead is much too high for in-memory column stores. *All* variants imply that the same tasks – and usually the same computations – are executed redundantly n times *and* that some sort of *reliable voter* compares the n different results. This always introduces the new problem of guaranteeing that the voter itself is reliable. Furthermore, DMR alone allows only *detection* of errors upon mismatch between the two replicas, while TMR allows to correct errors by choosing the majority, i.e. 2-out-of-3. When two replicas contain the exact same error, TMR "corrects" to the wrong result and when all three replicas differ, TMR can still only detect the error, albeit introducing three-fold overheads, contradicting with Requirement $\mathcal{R}2$. Consequently, we disregard techniques like DMR or TMR.

2.5.1 Operating System-Level Techniques

On the OS level, Döbel *et al.* introduced Romain, a framework which provides hardware error detection and correction capabilities to applications in a transparent manner [43]. It basically supports arbitrary n-way modular redundancy by starting n distinct processes and is supposed to support legacy or proprietary third-party software on commercial-off-the-shelf (COTS) systems without recompilation. Not all application state is duplicated, as read-only memory regions are shared among all replicas. Overheads in run-time are reduced by comparing the processes' states only before their state is externalized, e.g. CPU exceptions, page faults, or system (OS) calls. Each process is halted until all replicas reach the same externalization point and then some select application state is compared to detect errors. While their benchmarks show runtime overheads of up to 20% for DMR and up to 30% for TMR, the total redundant *work* still amounts to at least 200% and 300%, respectively, due to the process replication. Furthermore, all all writable memory regions are *implicitly* replicated, as well. Together this overhead is infeasible for in-memory DMSs, as this greatly limits the amount of processing power and main memory, since typically most of the used main memory is *not* read-only. [170] present a survey on integrity checking techniques for storage [170]. They find that there are three most commonly used techniques. First, mirroring (n-way modular storage redundancy) simply replicates all data. Second, RAID parity uses XOR to compute M parity blocks of a set of N disk blocks, such that $N + M$ blocks are stored, and the loss or corruption of any M blocks can be repaired [170]. Third and finally, checksums are used to detect errors, where typically hash functions are used and special requirements like security are addressed by cryptographic hash functions [170]. These techniques, however, only assure that data on disks is free of corruption and it is implicitly assumed that all data in main memory is error-free, as well. With the assumption of having all business data in main memory and only working on that in-memory data, their considerations do not directly apply for us.

2.5.2 Compiler-Level Techniques

On the compiler level, several solutions were proposed. Shirvani *et al.* use software ECC for main memory data [168]. Oh and McCluskey proposed procedure duplication and argument duplication at source level [132]. They targeted at enabling software fault tolerance while minimizing energy utilization. Rebaudengo *et al.* developed a source-to-source pre-pass compiler, which additional generates fault detection code whilst using a high level programming language. This introduces $3\times$ to $5\times$ overhead and still exhibits undetected error rates between 1 to 5% [149]. Oh *et al.* proposed error detection by duplicated instructions (EDDI) [134] and control-flow checking by software signatures (CFCSS) [133] These are software-only approaches which modify programs during the compilation phase. EDDI works, as the name suggests, by duplicating instructions and, for that, it uses different registers and variables for the redundant instructions. This helps to detect errors e.g. in the CPU's Arithmetic Logic Unit (ALU), but not to protect e.g. the execution path (branches), which is what CFCSS aims at. It assigns unique signatures to all nodes in a program graph and adds appropriate instructions for error detection. The signatures are computed and embedded during program compilation. Both approaches are too inefficient with regard to in-memory data management. Reis *et al.* introduced software-implemented fault tolerance (SWIFT), which introduces several refinements to EDDI and implements a software-only signature-based control-flow checking scheme [152]. While EDDI strives to protect memory data structures as well, SWIFT assumes that these are well protected by hardware mechanisms like parity or ECC. While the compiler techniques may be sensible solutions for legacy or non-database software, they alone are not sufficient to meet our Requirements $\mathcal{R}1$ to $\mathcal{R}5$. On the one hand, EDDI reduces overhead by exploiting instruction-level parallelism (ILP), but since DMSs do the same to minimize query latency, these are contradicting techniques. Further, performance improvements as in SWIFT can only be achieved when the data can already be assumed to be reliably stored. This is not the case for us, as we discussed earlier in this chapter. On the other hand, we do not consider protection of the control flow itself in this thesis, which is due to several reasons. First, we assume that memory cells are much more vulnerable to bit flips than others and that business data (non-executable memory regions) will be several orders of magnitude larger than executable memory regions and thereby consume virtually all main memory. Second, in compliance with

Requirement $\mathcal{R}5$, we seek separation of concerns and want to concentrate on protecting data and data structures. Techniques like SWIFT could be used to protect the control flow.

2.5.3 Application-Level Techniques

Shirvani *et al.* consider a separate task for error detection and correction (EDAC), which periodically sweeps certain main memory regions of a system, which were previously registered by a program for sweeping [168]. They use a block-oriented error code which is agnostic to the actual data stored and processed. Their sweep task, called EDAC program, has high priority, pauses other tasks and must complete one memory sweep before any other task resumes. They integrate two separate programs which can even cross-check each other to detect bit flips in the EDAC programs themselves. Their main interest is in protecting the code regions containing executable code, because they observed that machine code of the OS itself as well as applications is often enough damaged by bit flips, such that the system must be reset regularly. Nonetheless, their technique can be used to protect data, too. An application must explicitly ask the EDAC program to sweep its memory regions through inter-process communication. This work is evaluated in a real space satellite, where they have a bit flip-hardened system and a COTS system side-by-side, whereas the former is used to evaluate execution of the latter. They show that on hardware which is not specially hardened against the higher radiation in space, their technique considerable improves the reliability of the system. However, their approach contradicts with our Requirement $\mathcal{R}4$ in the sense that a program is paused during a memory sweep. While this time may be short for a satellite system with a very restricted amount of memory, this is not feasible for multi-terabyte systems for in-memory DMSs, where a complete sweep may take the time of many queries. Furthermore, their approach being agnostic to the actual data types and structures might keep from choosing the most effective (or efficient) error code, which is against Requirements $\mathcal{R}1$ and $\mathcal{R}2$.

In the database domain, there is also some related work, however not directly targeted towards the handling of arbitrary, transient bit flips in hardware. Lots of related work considers certain kinds of software bugs which lead to data corruption and mostly for disk-based DMSs instead of main memory-centric ones.

Pittelli and Garcia-Molina investigated the use of a triple modular redundant setup where a database is replicated across three computers, all of them executing the same transactions in the same order [138]. Their aim is to protect against arbitrary hardware faults in one of the three replicas, while they argue that their approach can be easily extended to n-modular redundancy [138]. Their definition of a failure is simplified to the fact that a database page differs from the majority of the replicas of this page. During operation, a query is distributed to and executed by all replicas in the same order. The results from all replicas are sent to the user and signatures of each result are generated and sent to voters, one on each node, to check for inconsistencies between the replicas. Since no additional error detection mechanisms are explicitly added, the authors state themselves that error detection can take a while and transaction may even work on corrupt data [138]. A voter only checks for corruption of the local replica and when it detects one, it tries to reconstruct a valid database with the help of the other nodes. Pittelli and Garcia-Molina use a two-phase operation where first the failing node asks the other replicas for consistent copies and then installs them in the second phase. To detect corruption, signatures ares used which shall represent (parts of) the whole database in an efficient manner [138]. Pittelli and Garcia-Molina use a hierarchical four-byte checksum, where the one representing the whole database is constructed from each of the page checksums. For detecting errors only the small database-wide checksum is exchanged and when an error is detected in one node, the large set of page checksums is transferred to detect the faulty pages. Afterwards, the respective pages are transferred to rebuild a consistent database snapshot. On the one hand, heir approach satisfies our Requirement $\mathcal{R}4$, as they do not halt the valid nodes during recovery of the faulty node. On the other hand, network messaging across nodes incurs high latencies. While their hierarchical checksum approach seems fast and viable, they do not provide measurements for TMR versus unprotected query runtimes. Since all results are returned to the user, it seems that the voting must be done at the user again. When the results are only returned after the voting, this would lead to higher latencies.

Shortly after, [177] introduced a method for protecting critical DMS data structures against software errors [177]. In their context, this may happen due to extensibility, where client code in the form of user-defined operators, access methods, etc. is executed inside the DMS. They use operating system support to write-protect certain memory regions, to prevent buffer overruns, stray writes, and the like. [177] present three variations, with the first one unguarding on a per-record basis (the respective memory page) before a record update and regarding afterwards. Each guard and unguard operation requires a system call, which can be very costly [177]. The second variant is called deferred write update model, where all records are left guarded until the end of transaction. For updates, records are copied to writable memory regions and only modified there. Another system call allows then to copy the modified copies into their respective places at the end of a transaction. This includes first unguarding the respective regions and then guarding them again. The third variant is called expose segment update model, where all pages are guarded and unguarded at once. As they show, protecting all data structures and the whole buffer pool of an in-memory database leads to query runtime overheads between 70% and 116% [177]. This work is not suited for protection against transient hardware errors, because bit flips are not induced by that sort of software bugs considered here. Even in write-protected pages, bit flip can happen and especially those bits which denote a page being protected could flip. Furthermore, protection overheads of more than 100% are undesirable, contradicting Requirement $\mathcal{R}2$.

Bohannon *et al.* could still assume that hardware was becoming more reliable in the early 2000's and dealt with software errors, such as the “addressing error”, which includes buffer overflows and “wild writes” [19]. They consider direct corruption due to addressing errors and indirect corruption due to processes working with already corrupted data and thereby writing corrupt data again. To detect such unintended modifications, they introduce a special interface for modifying data, which additionally takes care of so-called code words, which cover a whole protection region. They describe several requirements to the code word scheme and argue that several code word schemes would satisfy them, but effectively use a simple XOR of all words in the protection region [19]. Bohannon *et al.* introduce three corruption detection techniques, where the first, read prechecking, requires latches for serializing access to the code word (checksum). There, for reading the checksum is recomputed and validated against the stored one, however the authors do not note whether this is done in advance or concurrently to reading. For updates, exclusive latches are acquired during updates of the checksum. This variant shall prevent using corrupt data. The next one, data code word, shall detect direct corruption, by postponing the checksum comparison to a periodic, asynchronous maintenance operation, which they call audit. Their third variant, data code word with deferred maintenance, tries to further reduce contention on latches by storing code word updates in a log and updating the actual checksum during log flushes. One problem with checksum-based approaches in general is the chance that when data is read several times to compute the checksums for detection and updates, then in between these passes an arbitrary bit flip is not detected. Furthermore, as we will discuss in more detail in Section 3.2.2, XOR checksums can exhibit poor error detection performance compared to other coding schemes. Unfortunately, Bohannon *et al.* do not introduce their special interface, so that we can not evaluate whether it contains more flaws. Their performance evaluation shows that for a read-and-update workload, their methods lead to throughput overheads (in operations per second) between 8.5% and 22.4%

on favorable sizes of the protection region.

Also in security-related research did we find some related work regarding the "corruption" of data. Barbará *et al.* examine the case where an intruder wants to replace the latest data with old one *on disk*, for personal gain or to corrupt a service [15]. They assume that the intruder already gained knowledge about the system's internal structure and architecture and thus has the ability to store old data and protection information. To detect data corruptions, they use two levels of checksums over large blocks of data. The first level involves computing one checksum per data block and table attribute, while the second level computes checksums for several combinations of first-level checksums. In essence, Barbará *et al.* assume that the only unknown to the intruder are the secret keys for generating the checksums and they argue that *something* must be secure to build upon. Each change of a data item requires a chain of updates to those second-level checksums which cover the respective first-level checksum. As a consequence, to successfully inject old data without triggering a checksum mismatch on both levels, an intruder must overwrite all respective data blocks such that all the checksums on both levels match [15]. For our scenario, this is insufficient. First, they assume disk-based database systems, where the I/O costs can mitigate the costs of additional computations of checksums. Second, we do not assume a malicious adversary at the disk level, but arbitrary transient hardware errors (especially in main memory). While their technique detects errors which were induced while data was on disk, they cannot detect hardware errors. For instance, assume that a transaction changes a data item of one disk block and assume that some hardware error(s) silently corrupted data in main memory, then the checksums will be computed on that non-maliciously changed data. This is, because only data loaded from disk to memory is not trusted, but data in main memory is trusted, which clearly excludes the setting of this thesis.

From our perspective, the presented contemporary, general-purpose, software-based fault-tolerance techniques are either not appropriate to handle bit flips and protect our most precious good – the business data – or introduce too high overheads in terms of memory consumption, computing work, or both. Optimizations for reducing the run-time or redundant work overhead assume that data errors are detected by other means, which is not feasible in our case. Further, these techniques exhibit rates for undetected errors, which we assume unacceptable in our domain. This leaves the most crucial part of what reliable data management should store and process: *data*. The only conclusion we can draw is, that DMSs have to take care of handling bit flips in data and data structures themselves, in the application layer.

2.6 SUMMARY AND CONCLUSIONS

Reliable data storage and processing is the key objective of data management, in general. As we first discussed, the RCB could mask and hide most if not all errors in the lower hardware / software stack from applications. By that, DMSs could assume reliable hardware and operating systems, except mass storage devices. In the database domain, three failure classes concerning physical failures have been considered so far, ranging from system failures, over media failures to single page failures. The second most important requirement for DMSs is *efficient* storage and processing, which is achieved in part by adapting to modern hardware features. It is now widely acknowledged that hardware becomes less and less reliable due to the constant reduction of transistors feature sizes and accompanying negative side effects on the physical layer. Bit flip rates are exponentially increasing and bit flip weights exceed the error detection capabilities of error detection schemes employed today. Consequently, the RCB is reduced – more components become unreliable and less errors are masked. An additional challenge is that the error behavior changes at run-time, due to hardware aging effects. By that, the key requirement of data management – *reliability* – can not be met on future hardware anymore and the speed improvements are no longer a free ride. The database community could neglect these issues so far and, to the best of our knowledge, since several decades we take a first step towards handling them in the DMS itself by introducing bit flip *detection*. This is a very important step, because it is first and foremost crucial to detect that a bit flip occurred and, by that, to avoid *silent data corruption*. We formulated Requirements $\mathcal{R}1$ to $\mathcal{R}5$ for data management-specific solutions under bit flips, for effectiveness, efficiency, adaptability, availability, and separation of concerns. Within this thesis, we regard the following hardware components being affected by bit flips: 1. data caches in the CPU, 2. main memory, and 3. interconnects between these. Most of the related work used some form of n-modular redundancy of space, time, resources, or a combination of these. We believe that data management systems must handle bit flips themselves in software, because today's solutions are either inappropriate or too costly, with regard to our requirements. To meet our set of requirements for multi-bit flip detection, we will examine error control codes as an alternative to n-modular redundancy.

3

ANALYSIS OF CODING TECHNIQUES

The aim of this chapter is to select and evaluate error control codes suitable for storage and processing in main memory-centric DMSs. Therefore, we first provide an overview of error control codes and then select three potential candidates in Section 3.1. We examine the selected ones in greater detail regarding their coding operations (Sections 3.1.1 to 3.1.3) and their ability to detect errors (Section 3.2). We show that AN codes exhibit very beneficial properties regarding storage and processing of integers. We then compare the error codes regarding our Requirements $\mathcal{R}1$ to $\mathcal{R}3$ – effectiveness, efficiency, and adaptability – in Section 3.4. There, we show grave disadvantages of AN codes in terms of processing overheads. Afterwards, we present a fundamental improvement to AN coding in Section 3.5, which was not proposed before, and which greatly improves the efficiency of error detection and decoding. Finally, Section 3.6 summarizes this chapter.

The main contributions of this chapter are the following:

1. We investigate the use of error control codes in data management systems.
2. We introduce the terms *data hardening* and *data softening* to distinguish the purpose of error coding from the type of coding used by other domains like data compression or security. We will give two definitions which build upon each other. The first set of definitions will be given shortly, while the second set will be provided later in this chapter in Section 3.5.4.
3. With respect to Requirements $\mathcal{R}1$ to $\mathcal{R}3$, we select a few error control codes which could be employed for software-based fault-tolerance in main memory-centric DMSs. Part of this discussion was published in [87].
4. We provide a detailed analysis of the silent data corruption probabilities of the selected error codes.
5. To the best of our knowledge, we are the first to propose improvements for detection and decoding of AN coding, which greatly increase decoding and error detection throughput and which enable vectorization using SIMD instructions. This was published in [87].

3.1 SELECTION OF ERROR CODES

Error control coding is based on information theory, which was established by Shannon in 1948. Shortly after, Hamming and Golay developed the first applicable error control codes [57, 66]. We will use "error code" or just "code" as an abbreviation for error control code throughout this thesis. Since then, many code families were found and we will briefly introduce a few of them and discuss their applicability in our context. There is a rich amount of literature on this topic, including many standard text books such as [25, 109, 122]. Error coding adds redundancy to information to detect or even correct errors that corrupt data during transmission or storage, as depicted in Figure 3.1. An error code maps (encodes) a data word from the data domain onto a code word from the code domain. Only the code words in this mapping are considered *valid*. Since the code adds redundancy, there are other possible code words which do not belong to the mapping and are considered *invalid*. An important metric for error codes is the *Hamming distance* d_{H}, which denotes the number of symbols in which two code words differ. In our case it suffices to assume that a symbol is a single bit. An error in transmission or storage may transform a valid code word into an invalid one and there are codes which allow to correct invalid code words to the original data word.

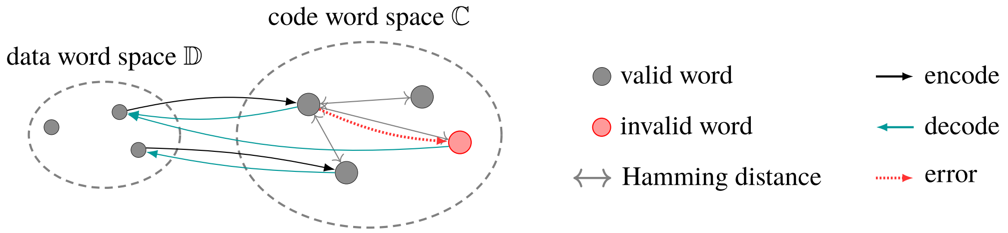


Figure 3.1: Basic graph representation of error coding.

There are other domains like data compression and security, which also use certain *data coding* operations for changing the data representation into a desired, different form. Before we proceed with the introduction of error coding schemes, we introduce new terms to distinguish from these other domains. For instance, in the domain of compression, those coding operations are called *compression* and *decompression* for mapping into another data representation and back, respectively. Regarding security, especially in the cryptography domain, there are operations to *encrypt* and *decrypt*[1] data to protect it from malicious modifications. Both domains, compression [4] and security [169], play important roles in the data management domain, as well. In our case, we want to use data coding to *detect* bit flips which occur due to *hardware errors*, and not due to some malicious or otherwise intended action. Consequently, to distinguish this area, we give new terms for our encoding and decoding operations.

Definition 5 (Data Hardening). *We denote the process of* encoding *data as* ***data hardening****, since data is literally firmed so that corruption in the form of bit flips becomes* detectable.

Definition 6 (Data Softening). *We denote as* ***data softening*** *the* decoding *of data, as it becomes (more)* vulnerable *to corruption in the form of bit flips again.*

Several dimensions of classifying error codes exist [25, 109, 122]:

- Based on how redundancy is added, codes are divided into *block codes* and *convolutional codes*, both being widely employed. For instance, block codes are used in server-grade ECC DRAM memory modules and convolutional codes are used for deep space transmission [114], just to name two examples.
- Based on channel error type, there can be codes for *independent error control* or for *burst errors*.
- Referring the error control strategy, there are error *detecting* and error *correcting* codes.
- Based on the possibility of separating the information symbols from the redundant ones, there are *systematic* codes, which contain the original data symbols in clear. In contrast, *non-systematic* codes do not contain them in clear so that data and code bits cannot be separated.

Block Codes are codes where the total coded information can be divided into blocks of n symbols. We here assume that such a symbol is a bit representing either 0 or 1, i.e. the *Galois Field* GF(2). The blocks are called *code words* and n is the *code length*, while the *data length* is

[1] We are well aware that data can be cryptographically *signed* as well, to detect malicious modification, but we want to show analogies between the operation of mapping from and to another domain.

denoted as k. The important property here is that each code word can be *decoded individually*. For block codes, typically the notation (n, k, d_H) is given, which presents the most important characteristics of the code. Block codes can further be distinguished being *linear* or *non-linear*. For a linear code, each *linear combination* of any two code words results in another code word, where linear combination is the binary addition without carries of two code symbols. The first error codes were linear block codes and the very first ones being Hamming codes [66] of which a single error correcting, double error detecting variant is still widely used. For now it suffices to know that code words are generated by multiplying a data word vector against a code generator matrix, which produces a code word vector. Generally speaking, the code bits are parity check bits over different sets of the data bits. While Hamming codes originally only have code word lengths $n = 2^m - 1$, shortening allows arbitrary data word lengths to be used. Shortly after, Golay codes were introduced [57], which have a fixed data and code length of 12 and 23, respectively. In contrast to Hamming codes, the Golay code can correct all triple errors, but requires lookup tables for coding, which makes it impractical, because such tables need to be stored in main memory and can be corrupted as well. Reed-Muller codes [122, Chapter 2.3] are a family of codes where hardening works by evaluating a Boolean function over all possible values of an input word, which can again be expressed as multiplication with a generator matrix. The matrix consists of $k + 1$ rows, where the first row is the all-1-vector and the remaining $k - 1$ rows form all possible 2^{k-1} bit combinations. Since the generator matrix, or any of its permutations, does not contain the identity matrix, Reed-Muller codes are *non-systematic*. Decoding works by selecting for *each* code bit a subset of the 2^{n-k} parity-check equations and trying to correct errors in each code bit individually. From [122, Chapter 2.3] we conclude that this is only efficiently doable in hardware using majority-logic circuits. Interestingly, when appending an additional overall parity-check bit to Hamming codes, this results in a Reed-Muller code. *Binary cyclic codes* are a subclass of linear block codes and comprise several families of codes. They provide more structure than the previous codes which allows efficient implementation of hardening and softening [122, Chapter 3]. A linear block code is cyclic if and only if *every cyclic shift* of a code word is again a valid code word. A nice property is that all code words (seen as polynomials) are multiples of a single generator code word (polynomial). This makes it at least easier to create a generator matrix, whereas matrices for systematic or non-systematic codes can be found. BCH codes are one such family of cyclic codes. Original Hamming codes are also instances of binary cyclic codes [122, Chapter 3.1]. The parity-check matrix is obtained by cyclic shifts of a parity-check polynomial. In hardware, hardening and softening is realized using *cyclic shift registers*. *Cyclic Redundancy Check* (CRC) codes are another form of cyclic codes. Here, a potentially large data block is divided by a polynomial and its remainder is used as the error check symbol. CRC codes are typically used for error detection only [122, p. 45]. There are even more complex codes like Reed-Solomon (RS) codes, Turbo codes, Low-Density Parity-Check (LDPC) codes, which however are more complex to encode and decode. Turbo and LDPC codes are *iteratively decodable* codes, which means the decoding algorithm is iterated several times to improve the error correction performance. Most of the above codes require special hardware circuits to be implemented in an efficient manner, so that a software-based implementation is already out of the question. Non-systematic block codes are also unfavorable, because they require to decode whenever we want to access the data, especially for frequent operations in DMSs like arithmetic and comparison operations. Hamming codes can be implemented relatively easy in software and as was shown they are instances of other code families, covering in fact a wide range of the types we briefly presented. Therefore, we will investigate Hamming codes in more detail after this general error code introduction.

Checksums are a further type of linear, systematic block codes. A checksum is a value computed from an arbitrary input sequence designed solely for error detection, where a potentially large input sequence is mapped to a comparatively small, usually fixed-sized value (with regard to the application). As the name suggests, the input sequence is somehow summed up to gain a check value. However, nowadays the term checksum is also used when actually hash functions are applied. There exists a multitude of algorithms with varying complexity and hardware support, e.g. parity bits, parity words, Message-Digest Algorithms (e.g. MD5) [156, 170] or cyclic redundancy checks (e.g. CRC32) [137, 170]. In the case of parity bits (words), the data bits (words) are summed up using the binary exclusive or operation (XOR, $\oplus$). Parity words are also referred to as Longitudinal range check (LRCID) [90]. The size of the resulting checksum can be arbitrary, but is usually either a single bit or aligned to machine words, respectively, for the sake of hardening throughput performance. In contrast to other checksums, parity words are one of the simplest methods to compute a checksum. Furthermore, XOR checksums can be computed using Single Instruction Multiple Data (SIMD) instructions of modern CPUs, which we will cover in detail later. Here it suffices to note that this is important for main memory-centric DMSs. Therefore, We will include the word-aligned *XOR* method into our further considerations. In the following we will simply use "XOR" as an abbreviation and we will call the computation of the checksum "hardening" as well.

Arithmetic error codes are a type of block codes whose unique feature is that arithmetic operations on code words result in valid code words again [13]. These are considered for protecting the arithmetic part (ALU) of processors, as well as transmission and storage of data. Avižienis suggests that the same code could also be used throughout a whole computer system, which could result in all data being hardened [13]. Avižienis calls this "*concurrent diagnosis* [. . .] *which occurs concurrently with the operation of the computer*" [13]. By that, transient and permanent faults can be detected without replicated storage or execution. This code class is very interesting due to its code-preserving property under arithmetic operations. One representative of arithmetic codes is the AN code, which has received some attention already some time ago [32, 41, 51] and we will investigate it in more detail, too. AN coding has gained lots of interest over the last decade, often used as a compiler technique to harden programs in total [49, 154, 155, 161] and in the embedded community [69], where it was more recently proposed for protecting controller execution in the automotive domain [55].

Convolutional Codes were invented shortly after the first block codes by Elias [45]. These are intended for infinite sequences of information symbols, where the contents of a code symbol depends also on previous information symbols. The code symbols do not necessarily have the same length. For instance, from the information sequence $i_1, i_2, i_3, \ldots$ the code sequence $i_1, i_1^{'}, i_2, i_2^{'}, i_3, i_3^{'}, \ldots$ is generated, where each $i_n^{'} = f(i_1, i_2, i_3, \ldots)$ is a function of the previous information symbols. Consequently, for convolutional coding, memory is required to store the last information symbols. Decoding happens by a kind of backtracing, an iterative process, where the decoder tries to find out what is the most likely input sequence which produced the generated message. Depending on how much time is available for decoding, the process can iterate several times and improve the decoding result. In contrast to block codes, this does not happen with a checker matrix, but using the *Viterbi algorithm* [180], which we do not sketch here, but which can be found in virtually all literature covering convolution codes, e.g. [25, 109]. The decoding process quickly incurs problems regarding computational and memory complexity. Furthermore, since the decoding depends on a history, or sequence, of code symbols, obtaining a single value is not trivial any longer. By that, we do not further regard this type of codes for employment in in-memory DMSs. We believe that the throughput performance implications are way too high as to do convolutional coding in software, which goes against Requirement $\mathcal{R}2$, even though the memory consumption might be smaller than for other codes with the same error detection capabilities.

To summarize the above, we distinguish coding operations regarding error detection from other domains by introducing the terms *hardening* and *softening* in Definitions 5 and 6, respectively. While there is a huge range of available error control codes, we will investigate in further detail the following three code families Hamming codes, XOR checksums, and AN codes. For the remainder of this thesis, we will need the following definitions:

Definition 7 (Bit Width). $|\mathbb{S}|$ *represents the largest effective bit width of any element in set* $\mathbb{S}$. *We will also call this the* bit width of set $\mathbb{S}$.

Definition 8 (Population Count). $[\![x]\!]$ *denotes the population count of x, which is the number of bits set to* 1 *in the binary representation of x.*

3.1.1 Hamming Coding

Hamming codes are originally known from telecommunications and systematically add parity bits over different sets of a data word's bits [66]. It is a family of block codes, meaning that one code is suited for a dedicated data block width, e.g. 64 bits. They are defined by the triple (n, k, d) which means its code words are n bits wide in total, its data (block) width is k bits and d denotes the *minimal Hamming distance* which will be explained in Section 3.2. Sometimes, also the short notation n/k is used. Hamming codes are employed in today's server-grade ECC main memory modules, where usually an additional 8 bits are stored for every 64 data bits, which is called a $72/64$ Hamming code, or $(72, 64, 4)$ in the (n, k, d)-notation. We will use the two notations interchangeably in the following. In theory, a code word c is obtained with the help of a generator matrix G multiplied with the data word (or message) m as

$$\begin{aligned} c &= mG, \\ G &= \big(I_k|P\big). \end{aligned} \tag{3.1}$$

Referring to the (n, k, d)-notation, Matrix G has n columns and k rows and it consists of the k-by-k identity matrix I_k and the *parity check sub-matrix* P. This form of G ensures the systematic property of codes. For each data width, there are potentially multiple different generator matrices. Although this may result in different codes, they have the same properties. For each such code there exists a parity check matrix H with n columns and $n - k$ rows. A received code word v can be validated by computing the *syndrome* S by

$$S = vH^T. \tag{3.2}$$

The syndrome vector S is the zero vector $\vec{0}$ if and only if v is a valid code word. Matrix G and H are formed such that their product results in the all-zero vector:

$$GH^T = \vec{0}. \tag{3.3}$$

For single bit flips, the syndrome directly indicates the position of the flipped bit. Assuming that G is in the systematic form, checker matrix H is of the form

$$H = \big(P^\top|I_{n-k}\big). \tag{3.4}$$

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

(a) Generator matrix.

$$H = \left(\begin{array}{cccc|ccc} 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right)$$

(b) Checker matrix.

Figure 3.2: The standard $7/4$ Hamming code example.

Consequently, it consists of the transposed checker sub-matrix from G (which has k columns and $n-k$ rows) and the $(n-k)$-by-$(n-k)$ identity matrix. The standard examples for the $(7,4,3)$ Hamming code are given in Figure 3.2, with the generator and the checker matrices shown in (a) and (b), respectively.

Extended Hamming Codes

In their basic form, Hamming codes have either only single error correction *or* double error detection capabilities (more details in Section 3.2.1). There is also an extended variant that adds an overall parity bit and can do both at the same time. The additional parity bit is generated over both all data and all other parity bits and increases the minimal Hamming distance of Hamming codes from 3 to 4 and also the total code width by 1. Extended Hamming codes are thus said to have SECDED capabilities. Extending codes in general is framed in [122, Chapter 6]. Originally, Hamming codes have fixed code word widths of $n = 2^m - 1$ bits where $m > 1$ and $k = n - m$, i.e., there are m parity bits. By that, for the original form there are $(127, 120, 3)$, $(63, 57, 3)$ etc. codes, while in the extended form there are $(128, 120, 4)$, $(64, 57, 4)$ etc. codes. In the following we will expect the *extended* form unless otherwise noted. An example for a $(7,4)$ Hamming code is given in Appendix A.2.1, Table A.5a.

Shortened Hamming Codes

To obtain a code with less *data* bits, e.g., a $(72, 64, 4)$ code[2], the $(128, 120, 4)$ code is shortened by using less data bits (64 instead of 120) and the same amount of parity bits. We define k_b as the basic Hamming code data width, k_s as the shortened Hamming code data width, and their difference as k_Δ by

$$k_\Delta = k_\mathrm{b} - \mathrm{s}. \tag{3.5}$$

Such codes can be constructed by removing the lower k_Δ rows in the generator matrix G and all resulting k_Δ zero-columns. The latter removes those columns which would just project the now removed data bits. For shortened (binary cyclic) codes such as Hamming codes, the same encoder and decoder can be used [122, p. 45]. Shortening codes in general is framed in [122, Chapter 6]. Examples of shortening codes are given in Appendix A.2.1, Tables A.5b and A.5c.

[2] Such a code was first proposed for the IBM model 360 mainframe [71] and is used in server-grade ECC DRAM main memory.

Code Position Binary	12 1100_2	11 1011_2	10 1010_2	9 1001_2	8 1000_2	7 0111_2	6 0110_2	5 0101_2	4 0100_2	3 0011_2	2 0010_2	1 0001_2	Pattern
Data Position	8	7	6	5		4	3	2		1			
p_1		●		●		●		●		●		●	$\texttt{0x5B}_{16}$
p_2		●	●			●	●			●	●		$\texttt{0x6D}_{16}$
p_3	●					●	●	●	●				$\texttt{0x8E}_{16}$
p_4	●	●	●	●	●								$\texttt{0xF0}_{16}$
p_{all}	●	●	●	●	●	●	●	●	●	●	●	●	(*)

Figure 3.3: Pattern for the parity bits of the shortened Extended 13/8 Hamming Code. The i$^{\text{th}}$ parity bit p_i computes the parity over all *data* bits whose binary representation of the code position has the i$^{\text{th}}$ least significant bit (LSB) set to 1. The right-most column shows the AND-ing pattern for each parity bit. (*): The overall parity p_{all} extends a 12/8 to a 13/8 Hamming code and includes all data bits and also the other parity bits.

	64	56	48	40	32	24	16	8	Pattern
p_1	● ● ● ●●	● ● ● ●	● ● ● ●	● ● ● ●	● ● ●●	● ● ● ●	● ● ●● ●	● ●● ●●	`0xAB 55 55 55 56 AA AD 5B`
p_2	●● ●● ●	● ●● ●	● ●● ●	● ●● ●	● ●● ●●	●● ●●	●● ●●	●● ●● ●	`0xCD 99 99 99 9B 33 36 6D`
p_3	●●●●	●●●● ●	●●● ●	●●● ●	●●● ●●	●● ●●	●● ●●●	● ●●●	`0x78 F1 E1 E1 E3 C3 C7 8E`
p_4	●	●●●●●●●	●	●●●●●●●	●●	●●●●●●	●●●	●●●●	`0x01 FE 01 FE 03 FC 07 F0`
p_5	●	●●●●●●●●	●●●●●●●		●●	●●●●●●●●	●●●●●		`0x01 FF FE 00 03 FF F8 00`
p_6	●	●●●●●●●●	●●●●●●●●	●●●●●●●●	●●●●●●				`0x01 FF FF FF FC 00 00 00`
p_7	●●●●●●●								`0xFE 00 00 00 00 00 00 00`

Figure 3.4: Pattern for the 72/64 shortened Extended Hamming Code. All shorter codes, like 39/32 or 13/8 (cf. Figure 3.3), can be derived from this scheme by removing the appropriate leading parts and removing parity bits with empty remaining lines. The overall parity bit (p_{all}) is not shown here as it only builds the parity across all data and parity bits.

Practical Construction

Practically, we need to neither generate matrices nor do matrix multiplication at all, since the code bits are actually parity check bits over different sets of data bits. The basic patterns for selecting the data bits are shown in detail for a $13/8$ code in Figure 3.3 and condensed for a $72/64$ code in Figure 3.4. Both are shortened Extended Hamming codes. For each parity bit, the data bits are selected by a fixed pattern, i.e. each is computed by AND-ing a pattern with the data word, followed by counting the one-bits (population count) and producing an even parity. For shortened (Extended) Hamming codes, the appropriate $k_\triangle$ columns are removed and also the patterns are appropriately shortened by $k_\triangle$ bits. From Figure 3.4, other shortened codes can be derived by removing further columns and parity bits when they do not cover any data subsets anymore.

3.1.2 XOR Checksums

XOR checksums are, like Hamming codes, systematic and linear codes. In general, parity bits are generated over a *block* of data items. We denote the set of all possible data items as $\mathbb{D}$, the number of these items over which a checksum is generated as $\#\mathbb{D}$, and the set of possible checksums is $\mathbb{X}$. The bit width of the data items $|\mathbb{D}|$ is not necessarily equal to the bit width of the checksum $|\mathbb{X}|$. A code word consists of

$$\begin{aligned} k &= \#\mathbb{D} \cdot |\mathbb{D}| \\ n &= k + |\mathbb{X}| = \#\mathbb{D} \cdot |\mathbb{D}| + |\mathbb{X}| \end{aligned}$$

bits. We can also describe the construction of XOR checksums with the help of a generator matrix exactly as in Equation (3.1). For parity *bits*, the generator matrix G is defined as

$$\begin{aligned} c^{\text{bit}}_{\text{XOR}} &= \big(d_1\ d_2\ d_3\ \dots p\big) = dG^{\text{bit}}_{\text{XOR}} \\ &= \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \end{pmatrix}^T \left(\begin{array}{cccc|c} 1 & 0 & 0 & \cdots & 1 \\ 0 & 1 & 0 & \cdots & 1 \\ 0 & 0 & 1 & \cdots & 1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \end{array}\right) \\ &= \big(d_1\ d_2\ d_3\ \cdots \mid p\big) \end{aligned}$$

(a) Parity bit.

$$\begin{aligned} c^{\text{word}}_{\text{XOR}} &= \big(c_1\ c_2\ c_3\ \cdots\big) = dG^{\text{word}}_{\text{XOR}} \\ &= \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \end{pmatrix}^T \left(\begin{array}{cccc|cc} 1 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 1 & \cdots & 1 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \end{array}\right) \\ &= \big(d_1\ d_2\ d_3\ \cdots \mid p_1\ p_2\big) \end{aligned}$$

(b) 2-bit word.

Figure 3.5: XOR generator matrix examples.

$$G^{\text{bit}}_{\text{XOR}} = (I_k|\vec{1_k}), \tag{3.6}$$

where I_k is the k-by-k identity matrix and $P = \vec{1_k}$ is a vector with k elements, all set to 1. Vector $\vec{1_k}$ has the effect that all bits in the data word are added together, which for binary addition is carry-less addition in GF(2) and this is the same as XOR-ing them together. The resulting code word $c^{\text{bit}}_{\text{XOR}}$ consists of all data bits d_i and the parity bit p. For parity *words*, the generator matrix is defined as

$$G^{\text{word}}_{\text{XOR}} = (I_k|V_1 \dots V_{|\mathbb{X}|}). \tag{3.7}$$

Each of the vectors V_i generates the parity over the i-th bit of each $|\mathbb{X}|$-wide block. Therefore, V_i has elements $i, i + |\mathbb{X}|, i + 2|\mathbb{X}|, \dots$ set to one and all others set to zero. Examples of XOR generator matrices are given in Figure 3.5 for parity bit (a) and 2-bit word (b) checksums, which can be easily extended to other word widths. The parity-check matrix H_{XOR} is constituted of the second part of the generator matrix, i.e. only the parity-generating columns (V_1) or $(V_1, \dots V_{|\mathbb{X}|})$, with the identity matrix below (in transposed form), which selects the appropriate parity bit for checking:

$$H^{\text{bit}}_{\text{XOR}} = \begin{pmatrix} \vec{1_k} \\ 1 \end{pmatrix}^\top = (\vec{1_n})^\top, \tag{3.8}$$

$$H^{\text{word}}_{\text{XOR}} = \begin{pmatrix} V_1 \dots V_{|\mathbb{X}|} \\ I_{|\mathbb{X}|} \end{pmatrix}^\top. \tag{3.9}$$

The check matrices are transposed to satisfy Equations (3.2) and (3.3). The syndrome S is computed the same as for Hamming codes as in Equation (3.2) and the code word is again only valid when $S = \vec{0}$ holds and Equation (3.3) holds likewise, too. In practice, the parities are computed bit-parallel in word-wise XOR operations and can be folded into the desired word-size at the end.

3.1.3 AN Coding

Arithmetic codes were first mentioned in the mid-1950s when Diamond proposed one form of AN coding [41]. Shortly after, his work was extended by Brown in 1960 [32]. Later, AN codes were developed for detecting errors in processors, mainly in the field of embedded systems [51, 161]. As we noted above, AN codes received quite some attention during the previous decade and was even proposed to be used in the automotive domain [55]. A code word $c \in \mathbb{C}$ is computed by multiplying a constant $A \in \mathbb{A}$ onto the data word $d \in \mathbb{D}$ (a concrete example is given below):

$$c = d \cdot A \tag{3.10}$$

$\mathbb{A}$, $\mathbb{D}_\Theta$, and $\mathbb{C}^A_{\mathbb{D}_\Theta}$ respectively are the sets of all possible parameters A, data words d of type Θ, and code words c obtained by multiplying the respective A onto all $d \in \mathbb{D}_\Theta$. In contrast to the previous codes, the data type will later be important. Since AN coding only works on integers, the parameter set is a subset of the set of integers: $\mathbb{A} \subseteq \mathbb{Z} \setminus \{0, 1\}$. $A = 1$ makes no sense, because it is the neutral element for multiplication. As a result of the multiplication, the domain of code words $\mathbb{C}^A_{\mathbb{D}_\Theta}$ expands such that only multiples of A become valid code words, and all other values are considered non-code words. The choice of "good" As will be discussed in Section 3.2.3. In contrast to the previous two codes, the data itself is modified and the resulting code words are *non-systematic*. Due to the convolution of multiplications it is not trivial anymore to obtain the original data words and a division is required for decoding:

$$d = c/A \tag{3.11}$$

Errors are detected by testing the remainder of the division by A, which must be zero, otherwise the code word was corrupted (c_ε) e.g. by some bit flip weight b:

$$c \equiv 0 \pmod{A} \tag{3.12}$$

$$(c_\varepsilon = c \circ b) \not\equiv 0 \pmod{A} \tag{3.13}$$

where $\circ$ can be a binary operation like, e.g., XOR ($\oplus$), OR ($|$), or AND ($\&$), or an arithmetic operation like $+, -$ depending on the actual error model. A very useful feature of AN coding is the ability to operate directly on hardened data. In particular, due to the monotony of the multiplication, the following operations on two hardened operands yield the expected results:

$$c_1 \pm c_2 = (d_1 \cdot A) \pm (d_2 \cdot A) = (d_1 \pm d_2) \cdot A \tag{3.14}$$

$$c_1 \circ c_2 = (d_1 \cdot A) \circ (d_2 \cdot A) \overset{1/A}{\equiv} d_1 \circ d_2 \,, \quad \circ \in \{<, \leq, =, \dots\} \tag{3.15}$$

Care must be taken for operations like multiplication

$$\text{OK} \quad c_1 \cdot d_2 = d_1 \cdot d_2 \cdot A, \tag{3.16}$$

$$\text{BAD} \quad c_1 \cdot c_2 = (d_1 \cdot A) \cdot (d_2 \cdot A) = d_1 \cdot d_2 \cdot A^2, \tag{3.17}$$

$$\text{OK} \quad c_1 \cdot c_2 \Rightarrow c_1 \cdot {}^{c_2}\!/_{A} = d_1 \cdot d_2 \cdot A, \tag{3.18}$$

and division

$$\text{OK} \quad \frac{c_1}{d_2} = \frac{d_1 \cdot A}{d_2} = \frac{d_1}{d_2} \cdot A, \tag{3.19}$$

$$\text{BAD} \quad \frac{c_1}{c_2} = \frac{d_1 \cdot A}{d_2 \cdot A} = \frac{d_1}{d_2}, \tag{3.20}$$

$$\text{OK} \quad \frac{c_1}{c_2} \Rightarrow \frac{c_1}{c_2} \cdot A = \frac{d_1 \cdot A}{d_2 \cdot A} \cdot A = \frac{d_1}{d_2} \cdot A. \tag{3.21}$$

Equations (3.16) and (3.19) are valid operations using a hardened and a softened operand. In contrast, the result from Equation (3.17) be invalid due to the resulting A^2 and Equation (3.20) would produce a softened result. By that, for multiplication with 2 hardened operands, one operand must be divided by A, as in Equation (3.18)). For division the result must be multiplied by A, as in Equation (3.21), whereas the correct order of the additional multiplication is crucial. The division of the two code words must take precedence to prevent a possible integer overflow *and* to retain the correct integer division semantics. This is, because

$$(d_1 \cdot A) \div (d_2 \cdot A) \cdot A \neq (d_1 \cdot A) \div (d_2),$$

where $\div$ denotes integer division, which rounds down to the lower integer ($a \div b \equiv \lfloor {}^{a}\!/_{b} \rfloor$). As an example, consider $d_1 = 3$, $d_2 = 2$, and $A = 5$, where

$$\big((3 \cdot 5) \div (2\dot{5})\big) \cdot 5 = (15 \div 10)\dot{5} = 5 \neq 7 = 15 \div 2 = (3 \cdot 5) \div 2.$$

The following provides a concrete code example, which will also serve as a continuous example throughout this chapter:

Example 1. *Our continuous code example in decimal representation for $A = 233$:*

$$\begin{array}{crl}
\textit{Hardening} & 1,165 & = 5 \cdot 233 \\
\textit{Softening} & 5 & = 1,165/29 \\
\textit{Detection} & 1,165 & \equiv 0 \qquad (\text{mod } 233) \\
 & (1.164_{\varepsilon} = 1,165 \oplus_{XOR} 1) & \equiv 232 \qquad (\text{mod } 233) \\
\textit{Arithmetic} & 1,165 + 2,097 & = (5 \cdot 233) + (9 \cdot 233) = (5 + 9) \cdot 233 \\
 & & = 3,262 \\
\textit{Comparison} & 1,165 < 2,097 & = (5 \cdot 233) + (9 \cdot 233) \equiv 5 < 9
\end{array}$$

For detection, the second line represents a bit flip of the LSB in its binary representation.

AN Coding Derivatives

Diamond was the first one who mentioned AN codes as "Checking Codes", in 1955 [41]. These are of the form

$$c = a \cdot n + b,$$

$$n = (c - b)/a.$$

There, codes were investigated which would support addition of code words which would result in valid code words, again. These already include a *checksum* b, which is added on top of parameter a.

However, this was only mentioned in the correspondence part of the proceedings of the IRE [41]. Shortly after, Brown provided a much more detailed description [32]. First, codes of the form $An + B$ are considered. The problem here is, that while arithmetic operations on code words are possible, the signature must always be handled carefully. For instance, when adding two code words

$$c_1 + c_2 = An_1 + B + An_2 + B = A(n_1 + n_2) + 2B,$$

B is added up and must be treated accordingly. Furthermore, other arithmetic operations such as multiplication or division are much more complicated. Brown also considered codes with $b = 0$ and even error correction, but only for codes with a minimal Hamming distance of up to 3. Several decades later, Forin also describes extensions to add next to the static signature B_c (ANB-codes) also a timestamp D on top (ANBD-codes) [51]:

$$c = A \cdot d + B_c(+D) \text{ with } 0 < B_c < A.$$

There, decoding becomes more complicated and signature and timestamp must be known at compile time. This is not feasible for database systems with huge amounts of arbitrary data and arbitrary combinations of operations in query processing. The additional timestamp makes handling of arithmetic operations on code words even more complex than before.

Subsequent work by Schiffel allows to harden existing software binaries or to add hardening at compile time, where not all variables' states need to be known in advance [161]. This is called software encoded processing (SEP) and compiler encoded processing (CEP), respectively. The former, SEP, leads to runtime overheads between $2\times$ and $25\times$ [161, Section 7.4.2], whereas the latter introduces overheads between $2\% \dots 81\%$ for AN codes, however these measurements are done in a 10-MBit half-duplex network environment and denote the reduction of throughput in queries per second [161, Section 8.5.4]. Outside of that network setting, fully AN encoded benchmarks exhibit slowdown from $2\times$ to $75\times$. Furthermore, in her work she only describes hardening integers of size $|\mathbb{D}| \in \{1, 8, 16, 32\}$ bits and pointers, where the hardened values are always 64 bits large. For database systems this would incur way too much memory overhead, an aspect we will discuss in detail later in Section 4.1.3.

Ulbrich *et al.* use AN coding in the CoRed project for software-based fault-tolerance in mixed-criticality control applications [178]. They provide useful hints and discover pitfalls, e.g. when mapping the decimal arithmetic to binary arithmetic with fixed-width native data widths [69]: For instance, if an *invalid* code word is dividable by A without remainder, it is larger than the largest encodable value of the original data width $d_{\text{max}} \in \mathbb{D}$. Furthermore, Ulbrich *et al.* only consider the two fixed data widths of 8 and 16 bits and As of the same width, respectively, which leads to code word widths of 16 or 32 bits, respectively. Considering that some database systems use all data widths from 1 to 32 bits, their findings are too meager.

3.1.4 Summary and Conclusions

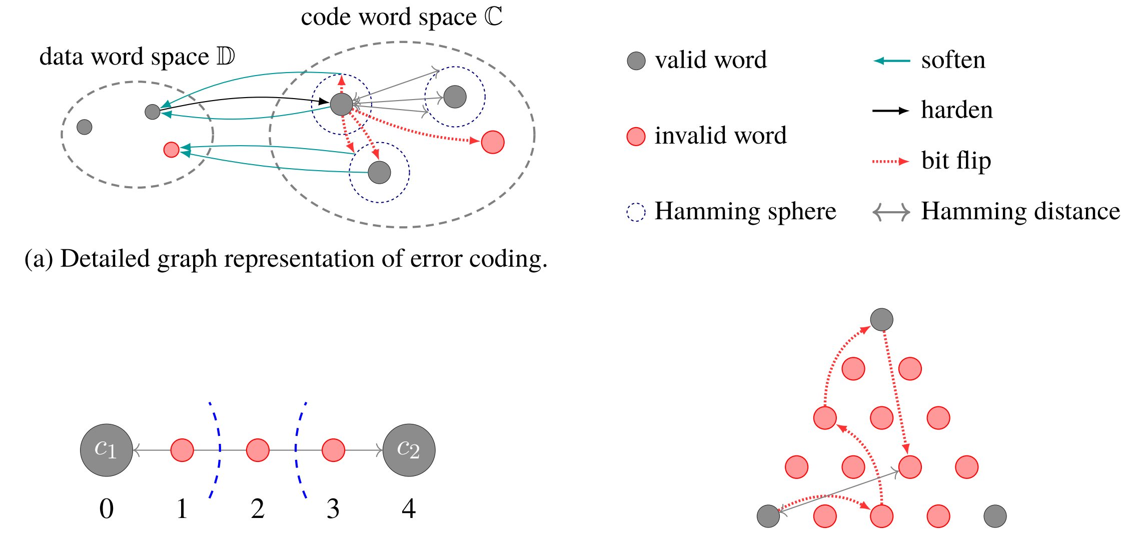
In this section, we discussed error control codes in general and chose three code families for further investigation to be used in main memory-centric DMSs. We started with introducing the terms hardening (encoding) and softening (decoding) for distinguishing from other coding domains like compression and security. Then, we gave a general overview of the huge range of error control codes, which can be classified using several metrics. Most importantly, we can distinguish between block codes and convolutional codes on the one hand, and between systematic and non-systematic ones on the other hand. We discussed a wide range of codes and we saw that some code families are specific instances of other code families. For instance, original Hamming codes are specific instances of BCH codes, while when appending an extra overall bit to Hamming codes, they become instances of Reed-Muller codes. We chose the three code families Hamming codes, XOR checksums, and AN codes to further investigate for use in main memory-centric DMSs. Then, in Sections 3.1.1 to 3.1.3 we presented the details of Hamming codes, XOR, and AN codes, respectively, regarding their basic coding operations and derivatives. For Hamming codes, we can use extended variants with an additional overall parity bit and we can shorten them to obtain a desired data width. Furthermore, Hamming codes can be computed using vectorized SIMD instructions, which we will discuss in detail later in Section 3.3. In theory, coding operations for both Hamming and XOR use matrix operations over data word vectors, but in practice today's CPUs support native instructions like population counting and bit-parallel XOR-ing. For these codes, error detection is done by recomputing the code bits and comparing those against the stored code bits. Using AN codes, hardening works my multiplying integers with a constant A, softening works via integer division, and error are detected by checking whether the remainder of the division by A is zero. A major advantage over the first two codes is the ability to directly execute arithmetic and comparison operations on AN code words. While there are also some derivatives for AN coding, we disregard them, because they make the coding operations much more complex and arithmetic operations cannot be done as easily any longer.

In order to be able to choose among these three codes, we need to take a closer look regarding our Requirements $\mathcal{R}1$ to $\mathcal{R}3$ for effectiveness, efficiency, and adaptability, respectively. Therefore, in the following Section 3.2, we will investigate the codes' probabilities of SDC, which means how good the codes can detect bit flips, which conforms to our Requirement $\mathcal{R}1$.

3.2 PROBABILITIES OF SILENT DATA CORRUPTION

No error code is perfect in the sense that it could detect *all* errors or bit flips. Consequently, the important performance criterion regarding *effectiveness* (Requirement $\mathcal{R}1$) of codes is the *probability* that they will *not* detect a bit flip in a code word. When this happens, an undetectable *decoding error* occurs and in the end corrupt data will be used. Figure 3.6a visualizes the problem at hand: Starting from the data word space $\mathbb{D}$, a code maps a data word into the code word space $\mathbb{C}$. From there we distinguish four cases:

1. In the best case, the code word stays unaltered and can be decoded back into the *original* data word.
2. A bit flip may keep the altered code word inside the original code word's *sphere of correction*. For Hamming, this is only the case for single bit flips. Then, the corrupted code word can still be decoded into the *original* data word. For error detecting codes like AN coding this case does not exist. Following the naming Hamming distance, we will call the sphere of correction *Hamming sphere*.
3. A bit flip may lead to such an invalid code word which differs so much from any other code word that it will be detected as invalid in any case. For Hamming, we know that this is the case for all double bit flips.
4. The last and worst case is when a bit flip leads to either directly another code word or into another code word's sphere of correction. This case cannot be detected and will decode into a *different* data word than was originally hardened, which is commonly called SDC.



(a) Detailed graph representation of error coding.

(b) Hamming distance d_H and Hamming sphere between two code words c_1 and c_2.

(c) Bit flip trace. Multiple corruptions of a code word are irrelevant as we are only interested in the state which is read from main memory or cache. The d_H between the original and last code word are important.

Figure 3.6: Concepts regarding the decoding error probability.

In general, to argue about the detection capabilities, we need to find the *distance distribution* of each code. For that, we need the *Hamming distance* d_H between all code words, which is indicated in Figure 3.6a by the gray arrows between the upper two code words. Figure 3.6b additionally shows the steps in between two arbitrary code words, where the Hamming distance between c_1 and c_2 is four ($d_H = 4$). To illustrate how to compute the actual probability of SDC, we model the set of all code words (valid and invalid) as a bidirectional, weighed graph, as in Figure 3.6a. There, each vertex represents a code word and each edge represent the transition (bit flip) which leads between each two code words, where The *edge weight* is equal to the Hamming distance. The Hamming distance, in turn, is the *Hamming weight* of the error pattern and is the population count of the corresponding bit flip. The distance distribution is the *histogram* of all edge weights between all *valid* code words. We only consider valid code words, because we assume that we intentionally only store valid ones. Since the graph is fully connected, a series of bit flips that actually occurs in reality can be modeled by the edge between the original code word and the one that is actually read. All intermediate states are not observed anyways and need not be considered. In Figure 3.6c a series of bit flips is shown on a code word and it shall emphasize that only the actual state which is read is of interest.

Typically, to keep things simple, a code is evaluated by its *minimal* Hamming distance $d_\mathrm{H,min}$, which is the smallest Hamming distance between *any* of a code's valid code words. $d_\mathrm{H,min}$ restricts the number of bit flips a code can correct. This restriction is the *Hamming sphere* around each code word, shown in Figures 3.6a and 3.6b as blue dashed circles around the code words (in Figure 3.6b it is only an arc). The Hamming sphere is the n-dimensional sphere which includes all invalid code words which can be corrected to the according nearest valid code word. All code words which are *not* included in *any* sphere can not be corrected, because there is no nearest valid code word. The Hamming sphere is constrained by the minimal Hamming distance $d_\mathrm{H,min}$, as its radius t is given by

$$t = \left\lfloor \frac{d_{\mathrm{H,min}} - 1}{2} \right\rfloor, \tag{3.22}$$

and each sphere contains

$$\sum_{i=1}^{t} \binom{n}{i} \tag{3.23}$$

invalid code words. This has an important implication: any code with $d_{\mathrm{H,min}} \geq 3$ can *in principal* correct some invalid code words, namely all invalid ones which are *inside* the Hamming sphere. There are *perfect* codes like the original (neither shortened nor extended) Hamming code where *all* invalid code words are inside the Hamming sphere of exactly one valid code word. The example code in Figure 3.6b has a minimal Hamming distance of four ($d_{\mathrm{H,min}} = 4$) so that the sphere radius is one. By that, the example code can correct *all* single bit flips and detect *all* double bit flips. It is an example of the (shortened) extended Hamming codes used in ECC DRAM. The Hamming spheres shown in Figure 3.6 are projections from the n-dimensional code space. While this metric may have been sufficient for former assumptions where only 1 or 2 bits flip, it will be too simple a metric for the future. First, larger numbers of bit flips are expected and, second, the bit flip weight is expected to increase over a system's life time (aging). By that, we also must consider the probabilities in how far a code can detect higher bit flip weights ($\geq d_{\mathrm{H,min}}$). To compute the probability of SDC for a given code $\mathbb{C}$, we need the actual distance distribution $K^{\mathbb{C}}$, defined as

$$K^{\mathbb{C}} = \{\kappa_b^{\mathbb{C}} \mid 1 \leq b \leq n = |\mathbb{C}|\}, \tag{3.24}$$

where we call $\kappa_b^{\mathbb{C}}$ the elements or *counts* of the distance distribution of *weight* b, for code $\mathbb{C}$. In other words, $\kappa_b^{\mathbb{C}}$ is the number of undetectable transitions between two valid code words of weight b. We also need the possible bit flip weight distribution, which is the upper bound for the number of possible b-bit flips that can happen. We call this bound $\beta_b^{\mathbb{C}}$ and it includes all the transitions from the 2^k valid code words (with $k = |\mathbb{D}|$) to all other valid and invalid code words of weight b:

$$\beta_b^{\mathbb{C}} = 2^k \cdot \binom{n}{b} \quad , 1 \leq b \leq n = |\mathbb{C}|. \tag{3.25}$$

The binomial coefficient is used because the order of how the bits flipping is irrelevant. Each code has exactly $2 \cdot \binom{2^k}{2}$ undetectable transitions, if there are no restrictions on the allowed sequence of code words. This is exactly the number of transitions between any two valid code words $c_1 \leftrightarrow c_2, c_1 \neq c_2$ and exactly the sum of all $\kappa_b^{\mathbb{C}}$. The factor 2 is required, because we count *two* transitions per pair, forth and back, and we can determine this sum in a general manner:

$$
\begin{aligned}
\sum K_{\text{det}}^{\mathbb{C}} &= \sum_{b=1}^{n} \kappa_b^{\mathbb{C}} \\
&= 2 \cdot \binom{2^k}{2} \\
&= 2 \cdot \frac{2^k!}{2! \cdot (2^k - 2)!} \\
&= 2 \cdot \frac{1 \cdot 2 \cdot 2 \cdot \ldots \cdot 2^k}{1 \cdot 2 \cdot [1 \cdot 2 \cdot \ldots \cdot (2^k - 2)]} \\
&= (2^k - 1) \cdot 2^k
\end{aligned} \tag{3.26}
$$

Equation (3.26) applies for error *detecting* codes only. For error correcting codes, we also need to consider the transitions into the correction sphere around each code word. Using Equations (3.22) and (3.23), we generalize Equation (3.26) to also include correcting codes:

$$
\sum K_t^{\mathbb{C}} = \sum_{i=0}^{t} \binom{n}{i} \cdot \sum K_{\text{det}}^{\mathbb{C}} = \sum_{i=0}^{t} \binom{n}{i} \cdot (2^k - 1) \cdot 2^k \tag{3.27}
$$

For detecting-only codes we set $t = 0$, resulting in

$$
\sum_{i=0}^{t=0} \binom{n}{i} = 1 \Rightarrow \sum K_{t=0}^{\mathbb{C}} = 1 \cdot (2^k - 1) \cdot 2^k.
$$

and for an SECDED code, we can set $t = 1$, which results in

$$
\sum_{i=0}^{t=1} \binom{n}{i} = 1 + n \Rightarrow \sum K_{t=1}^{\mathbb{C}} = (1 + n) \cdot (2^k - 1) \cdot 2^k.
$$

As a consequence, all codes with the same (n, k) parameters have the *exact same* number of undetectable bit flips. The different qualities arise from their different shapes of the distance distributions and whether a code is used for correction or not. We do not assume a specific error model, so consequently, we will restrict the following considerations to the *conditional* probabilities, i.e. the raw distance distributions. At the end of this section, we will show how to obtain unconditional probabilities by combining the conditional probabilities with two concrete error models: (1) equal bit flip probabilities, and (2) the binary symmetric channel (BSC). Anyhow, the *conditional* SDC probability is $P(\text{SDC}|b)^{\mathbb{C}}$ or for short $\varphi_b^{\mathbb{C}}$ for code $\mathbb{C}$, that a b-bit flip is not detected. This is the proportion of undetected b-bit flips with regard to all possible b-bit flips (the upper bound $\beta_b^{\mathbb{C}}$). It is computed as

$$
P(\text{SDC}|b)^{\mathbb{C}} = \varphi_b^{\mathbb{C}} = \frac{\kappa_b^{\mathbb{C}}}{\beta_b^{\mathbb{C}}} = \frac{\kappa_b^{\mathbb{C}}}{2^k \cdot \binom{n}{b}}. \tag{3.28}
$$

There, $\kappa_b^{\mathbb{C}}$ is the main coding-specific property that needs to be ascertained, whereas $\beta_b^{\mathbb{C}}$ can be computed just from the code properties n and k. We will now have a closer look at the three codes regarding their error detection capabilities. We will first revise from literature how to obtain them for Hamming codes. Afterwards, we will devise strategies for XOR checksums and AN codes. For both, there is to the best of our knowledge no *sufficient* description in the literature.

3.2.1 Probabilities of Hamming Codes

Since Hamming codes are linear codes, *every* code word is an error pattern which, when XOR-ed onto another valid code word, results again in a valid code word. Conversely, all error patterns that lead to undetectable bit flips are code words, i.e. both sets are identical. By that, the distance distribution is simply the histogram of the Hamming weights of all code words, also called the *weight distribution*

$$A_t^{\mathrm{H}} = |\{c \in \mathbb{C}_{\mathrm{H}} | w_{\mathrm{H}}(c) = t\}| \tag{3.29}$$

where $w_{\mathrm{H}}(c)$ is the Hamming weight of code word c and A_t is the number of code words in Hamming code $\mathbb{C}_{\mathrm{H}}$ with weight t. Consequently, for a binary linear code such as Hamming, the weight distribution and the distance distribution are the same. Fortunately. for each binary linear code in general, and each Hamming Code in particular, there exists a *weight enumerator* polynomial which directly tells us the weight distribution of that linear code. For plain n, k, d Hamming codes where $n \in \{2^{m-1} | m \in \mathbb{N}\}$ and $d = 3$, there exists a generic weight enumerator [120, Section 3.5, p. 98]:

$$A(\mathfrak{z})^{\mathrm{H}} = \frac{1}{n+1}\Big[(1+\mathfrak{z})^n + n(1-\mathfrak{z})(1-\mathfrak{z}^2)^{(n-1)/2}\Big]. \tag{3.30}$$

A few examples are given in [120], e.g. for the $(7, 4)$ Hamming code the weight enumerator is

$$A(\mathfrak{z})^{(7,4)} = \frac{1}{8}\Big[(1+\mathfrak{z})^7 + 7(1-\mathfrak{z})(1-\mathfrak{z}^2)^3\Big] = 1(\mathfrak{z}^0) + 7\mathfrak{z}^3 + 7\mathfrak{z}^4 + \mathfrak{z}^7. \tag{3.31}$$

This means there are 1 code word of weight 0 (obviously, the data word 0 results in all code bits 0), 7 code words of weight 3, 7 code words of weight 4, and 1 code word of weight 7 (all 1s). For the $(15, 11)$ Hamming code the weight enumerator is

$$\begin{aligned} A(\mathfrak{z})^{(15,11)} &= \frac{1}{16}\Big[(1+\mathfrak{z})^{15} + 15(1-\mathfrak{z})(1-\mathfrak{z}^2)^7\Big] \\ &= 1(\mathfrak{z}^0) + 35\mathfrak{z}^3 + 105\mathfrak{z}^4 + 168\mathfrak{z}^5 + 280\mathfrak{z}^6 + 435\mathfrak{z}^7 \\ &\quad + 435\mathfrak{z}^8 + 280\mathfrak{z}^9 + 168\mathfrak{z}^{10} + 105\mathfrak{z}^{11} + 35\mathfrak{z}^{12} + \mathfrak{z}^{15} \end{aligned} \tag{3.32}$$

We quickly see the symmetry of the coefficients: $1, 7, 7, 1$ for the former and $1, 35, 105, \ldots, 105, 35, 1$ for the latter example. We included $(\mathfrak{z}^0)$ which is essentially the all-zero code word, to show the symmetry, but it does not play any role for the further considerations on SDC, because the all-zero error does not actually change the code word. Now, we still need to obtain the distance distribution elements κ_b^{H} from the weight enumerator $A(\mathfrak{z})^{\mathrm{H}}$ for Equation (3.28), to obtain the conditional probability $\varphi_b^{\mathbb{C}}$. Since the set of code words is identical to the set of error patterns for any code word, to get the number of transitions from any valid code word to any other valid one, each enumerated weight must be multiplied by the number of *valid* code words 2^k (with $k = |\mathbb{D}|$):

$$\kappa_b^{\mathrm{H}} = 2^k \cdot A(\mathfrak{z})^{\mathrm{H}}[b], \tag{3.33}$$

where $[b]$ is the b-th component from weight enumerator $A(\mathfrak{z})^{\mathrm{H}}$, i.e. the factor of $\mathfrak{z}^b$. When plugged into Equation (3.28), this results in

$$\varphi_b^{\mathrm{H}} = \frac{\kappa_b^{\mathrm{H}}}{\beta_b} = \frac{2^k \cdot A(\mathfrak{z})^{\mathrm{H}}[b]}{2^k \cdot \binom{n}{b}} = \frac{A(\mathfrak{z})^{\mathrm{H}}[b]}{\binom{n}{b}}. \tag{3.34}$$

Consequently, to obtain the SDC probability we only need to compute (or count) the weight distribution and divide it by the appropriate binomial coefficient $\begin{pmatrix} n \\ b \end{pmatrix}$. Note that, since $A(b = 0)$ includes the all-zero code word, it follows that $\varphi_{b=0}^{\mathrm{H}} = 2^k$, which is the transition from each valid code word to itself and which is of course not an error. In other words, these transitions denote the non-bit flip case.

Extended Hamming Codes

From the above weight enumerators (Equations (3.30) to (3.32)) we can easily derive ones for the Extended Hamming codes which add the overall parity bit. We therefore add the factors of the $\mathfrak{z}$ with odd exponent to the ones with the next higher even exponent. This is, because the exponent of $\mathfrak{z}$ is the Hamming weight of the respective code words and the additional parity bit is 0 for even Hamming weights and 1 for odd ones. For instance, this will result in the enumerators

$$A(\mathfrak{z})^{(8,4)} = \mathfrak{z}^0 + 14\mathfrak{z}^4 + \mathfrak{z}^8 \tag{3.35}$$

$$A(\mathfrak{z})^{(16,11)} = \mathfrak{z}^0 + 140\mathfrak{z}^4 + 448\mathfrak{z}^6 + 870\mathfrak{z}^8 + 448\mathfrak{z}^{10}140\mathfrak{z}^{12} + \mathfrak{z}^{16} \tag{3.36}$$

for $(8, 4)$ and $(16, 11)$ Hamming codes, respectively. We can obtain the extended Hamming weight distribution $\kappa_b^{\mathrm{H}_{\mathrm{ext}}}$ directly from the original Hamming code's weight distribution by

$$\kappa_b^{\mathrm{H}_{\mathrm{ext}}} = \begin{cases} 0 & \text{, when } b \equiv 1 \mod 2 \\ \kappa_b^{\mathrm{H}} + \kappa_{b-1}^{\mathrm{H}} & \text{, otherwise} \end{cases} \tag{3.37}$$

Note that, for $b = 0$, there is no $\kappa_{b-1}^{\mathrm{H}}$ so that in this case this term is set to zero. We also want to emphasize that, the detecting-only Extended Hamming detects *all* odd error patterns due to the parity bit.

Correcting Extended Hamming Codes

Equations (3.30) to (3.37) only provide the error pattern distribution for *detection-only* Hamming codes. Due to correction, the 1-bit sphere must be taken into consideration (cf. Figure 3.6 and eq. (3.27)). Then, all transitions which lead to invalid code words which are 1 bit away from all other code words are also not detectable, because these are corrected to that next code word. An undetectable error pattern of weight b thus leads to additional $n - (b - 1)$ error patterns of weight $b + 1$, and additional $b + 1$ error patterns of weight $b - 1$ (note the mixed signs). Since

$\kappa_b^{\mathrm{H_{ext}}} = 0$ for any *odd* error pattern, only the numbers for these odd ones change, but not those of the even error patterns. Since for $b = n$ there is no $\kappa_{b+1}^{\mathrm{H_{ext}}}$, in this case that term is zero. The adapted enumerated weights $\kappa_b^{\mathrm{H^{cor}_{ext}}}$ of all the error patterns can thus be easily obtained by

$$\sigma^{\mathrm{H}}(x) = \kappa_b^{\mathrm{H^{cor}_{ext}}} = \begin{cases} \kappa_b^{\mathrm{H_{ext}}} & \text{, when } b \equiv 0 \mod 2 \\ \kappa_{b-1}^{\mathrm{H_{ext}}} \cdot \big(n - (b-1)\big) + \kappa_{b+1}^{\mathrm{H_{ext}}} \cdot (b+1) & \text{, otherwise} \end{cases} \tag{3.38}$$

Shortened Correcting Extended Hamming Codes

It is not as easy to obtain the error pattern weight distribution for multi-shortened Extended Hamming Codes, e.g. $(13, 8)$, $(22, 16)$, $(39, 32), \ldots$ codes [54, 97]. One would need them in database systems to naturally reflect the built-in data types of tiny, short, normal and big integers (8-, 16-, 32-, and 64-bit integers, respectively). There are also systems which put this to the extreme being tailored to work on any bit width from 1 to 32 bits [189]. A method that is sufficient in these cases is the brute force weight counting of all code words. It is easy to see that the required work of this brute force approach increases by each additional data bit with a factor of *two*, because now there are twice as many code words to determine the weight for. A possibility to reduce the complexity is to reuse previous codes' distributions. When an additional data bit does *not* require another parity bit in an $(n + 1, k + 1)$ Hamming code, we can trivially see that its weight distribution includes the one from the (n, k) Hamming code. Otherwise, when an additional data bit *does* require an additional parity bit, i.e. we have an $(n + 2, k + 1)$ code, then the additional parity bit only covers this new most significant data bit. Consequently, it is important only for those (new) code words that have the new most significant data bit set to one and we can see that the weight distribution from the (n, k) code is also included. By that, we only have to count the *additional* code words and we can reduce the complexity of the weight enumeration by a factor of 2. On our second evaluation system (cf. Table 3.7), for a $(47, 40)$ shortened correcting Extended Hamming code it takes about 10.5 minutes to compute the SDC probability of the reduced code word set and 21.1 minutes to compute the SDC probabilities of all codes up to and including that $(47, 40)$ code. Projecting the runtime up to the $(72, 64)$ code, counting the weight distribution would take approximately 335 years on a single machine for our second evaluation system.

In the following we will always only assume (shortened) correcting Extended Hamming codes. Figure 3.7 displays the distance distributions for the 4 codes (13,8), (22,16), (30,24), and (39,32). For each code, it shows both the upper bound β_b^{H} of possible bit flips as gray line, the code's distance distribution κ_b^{H} as bars (right x-axis), as well as the conditional SDC probability φ_b^{H} as colored line. First of all, since $d_{\mathrm{H,min}} = 4$, there are no undetectable errors for $b \in \{1, 2\}$. Second, there is a zigzag pattern in both the distance distribution and more so in the SDC, which is close to 1 for most of the odd b. The probabilities decrease for larger codes and for the (39,32) code, the high ones for the odd b are between 48 and 70%. This implies that for odd bit flip weights, chances of decoding error are very high. This is due to the correction capability of Extended Hamming codes, by which the 1-spheres (cf. Figure 3.6a) of all valid code words are included in the counted transitions. In particular, the additional overall parity bit makes all code words have an even weight so that only even bit flip weights lead to other valid code words. The 1-sphere adds the odd counts, so that without correction, κ_b^{H} would be zero for all odd b.

The zig-zag patterns shown in Figure 3.7 apply to all code widths. This indicates the probabilities of SDC of today's server-grade ECC main memory modules. Especially for the odd bit flip weights

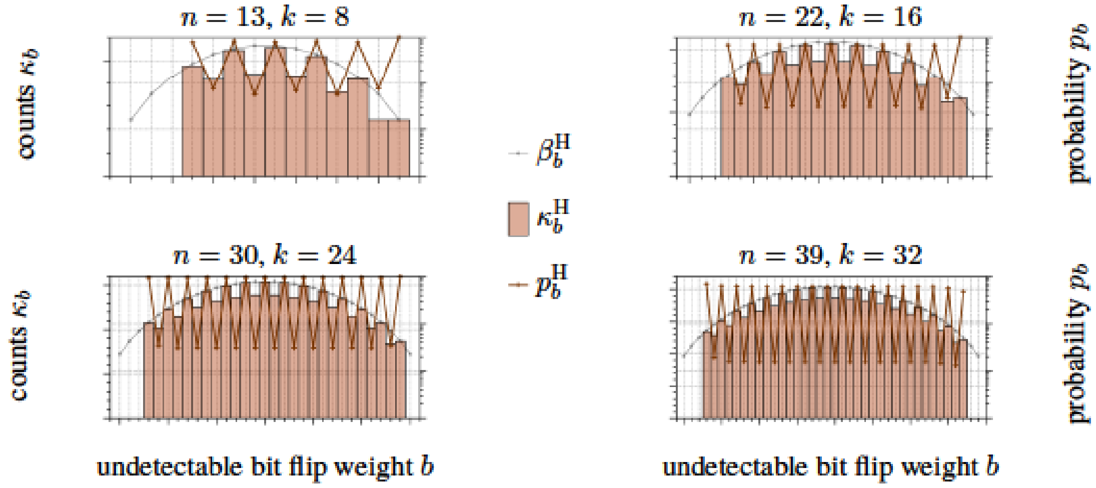


Figure 3.7: Conditional SDC probabilities for shortened Extended Hamming codes (13,8), (22,16), (30,24), (39,32) (H$\hat{=}$$\mathrm{H}_{\mathrm{ext}}^{\mathrm{cor}}$).

≥ 3, correcting Hamming codes exhibit very high decoding error (SDC) probabilities. Hamming codes themselves cannot be scaled to detect higher bit flip weights due to their structure, i.e. the fixed construction patterns.

3.2.2 Probabilities of XOR Checksums

Besides some very basic considerations as these by Maxino [115, 116], in the literature there is little to no detailed investigation of the whole range of possible silent data corruption probabilities. We will now show a way to succinctly compute the weight distribution of arbitrary XOR checksum codes. We derive methods to exactly compute the distributions for 1-, 2-, and 3-bit checksums and arbitrary block lengths. We assume that the block length is always a multiple of the checksum width, so that a block only contains complete data words. Determining the undetected error probability of checksums in general is at least as difficult as for Hamming codes, because it also depends on three parameters:

1. the actual checksum *type* (e.g. summation, XOR-ing, CRC, ...),
2. the checksum *width* (e.g. 16 bits for CRC-16, 32 bits for CRC-32, ...) and
3. the actual *block size*.

The first one (checksum type) is responsible to generate a code graph where all valid code words are as far away from each other as possible, with respect to the Hamming distance. The second (checksum width) defines the code's margin, i.e. the size of space of code words. Larger margins allow codes with larger minimal Hamming distance. The latter (block size) determines over how many data units the checksum is built. Changes in any of the three parameters lead to variation in the number of *possible* code words and variation in the distribution of *valid* code words, which results in varying detection properties. Considerations for summation checksums are discussed by Wakerly [181]. However, the only useful property derived there is that the minimum Hamming distance of such codes is two. By that, summation checksums can only guarantee to detect at least all single bit flips. We will now consider the case for XOR checksums.

Since XOR checksums are generated by simply XOR-ing all data elements with each other, any 2 bit flips in the exact same position, regarding the checksum width, cancel each other out. By that, the minimum Hamming distance of such codes is always 2. What can also be seen relatively easy is that *any odd-numbered* bit flip is detectable. Furthermore, the larger the data block over which the checksum is built, the larger the *total* chance of decoding errors, because there are more potential collisions.

Due to the XOR operation, XOR checksum codes are linear codes. By that, as for Hamming codes the distance distribution can be derived from the weight distribution and Equation (3.34) applies appropriately. However, the problem of computing or counting the weight distribution is still a hard problem and even harder than for Hamming codes. Again, with every *code bit* the effort doubles and since checksums operate on *blocks* of data, with every additional data word that is covered by a checksum, the time for counting multiplies by $2^{|\mathbb{D}|}$.

Weight Distribution Matrix

We made a first step towards computing *arbitrary* weight distributions for XOR checksums depending on a given checksum width and depending on a given number of data words protected by that checksum. Here, we assume that all data words have the same bit width as the checksum. To do so, we *counted* the weight distribution for increasing checksum widths $(1 \ldots 6)$ and increasing numbers of data words $(1 \ldots 8)$. As an example, for the 2-bit case the raw weight distributions are shown in Table 3.2. While we could not deduce formulas for any arbitrary checksum widths and any arbitrary code lengths, we found that grouping the distributions by the data weight (population count, $[\![\mathbb{D}]\!]$) and by the checksum weight ($[\![\mathbb{X}]\!]$) leads to important findings. Therefore, we define the (XOR checksum) *Weight Distribution Matrix* $\mathbb{W}_{\text{XOR}}^{\#\mathbb{D},|\mathbb{X}|}$ for a given number of data words ($\#\mathbb{D}$) and checksum width ($|\mathbb{X}|$) as:

$$\mathbb{W}_{\text{XOR}}^{\#\mathbb{D},|\mathbb{X}|} = \sum_{[\![\mathbb{D}]\!]=0}^{|\mathbb{D}|} \sum_{[\![\mathbb{X}]\!]=0}^{|\mathbb{X}|} \mathfrak{w}_{[\![\mathbb{D}]\!],[\![\mathbb{X}]\!]}^{\#\mathbb{D}} \tag{3.39}$$

Here, $\mathfrak{w}_{[\![\mathbb{D}]\!],[\![\mathbb{X}]\!]}^{\#\mathbb{D}}$ is, for a number of data words $\#\mathbb{D}$, the number of code words of *weight* $[\![\mathbb{D}]\!] + [\![\mathbb{X}]\!]$ grouped by data weight $[\![\mathbb{D}]\!]$ and checksum weight $[\![\mathbb{X}]\!]$. Examples are given in Table 3.1 for 1-bit checksums over increasing block lengths, i.e. numbers of data words $\#\mathbb{D}$. These matrices are an intermediate step towards computing the codes' weight distributions and there are several observations, which we verified for all counted weight distributions:

1. To obtain the weight distribution elements, the upward-right, colored diagonals' cross totals must be summed up (Counting Pattern matrices):
$$\kappa_b^{\text{XOR}}(\#\mathbb{D}, |\mathbb{X}|) = \sum_{[\![\mathbb{D}]\!]+[\![\mathbb{X}]\!]=b} \mathfrak{w}_{[\![\mathbb{D}]\!],[\![\mathbb{X}]\!]}^{\#\mathbb{D}}$$
2. Each weight distribution, in turn, represents the first column of the following matrix which covers one more data word. This means, when adding another data word to a checksum block, the previous weight distribution is fully included, because that new data word could be all-zero. Especially for the 1-bit case, the weight distribution for $d = \#\mathbb{D}$ numbers of data words equals the d-th line in Pascal's triangle, except that each *second* entry is set to zero.

$\lvert\mathbb{X}\rvert = 1$		Counting Pattern		Weight	Weight			Position Pattern		Next Rightmost	
		$[\![\mathbb{X}]\!]$		Row	Distribution		$[\![\mathbb{D}]\!]$	$[\![\mathbb{X}]\!]$		$[\![\mathbb{X}]\!]$	
$\#\mathbb{D}$	$[\![\mathbb{D}]\!]$	0	1	Sum	b	κ_b		0	1	0	1
1	0	1	0	1	0:	1	0	1	0	1	0
	1	0	1	1	1:	0	1	0	1	0	1
Col Sum		1	1		2:	1					
2	0	1	0	1	0:	1	0	1	0	1	0
	1	0	2	2	1:	0	1	0	2	0	2
	2	1	0	1	2:	3	2	1	0	1	0
Col Sum		2	2		3:	0					
3	0	1	0	1	0:	1	0	1	0	1	0
	1	0	3	3	1:	0	1	0	3	0	3
	2	3	0	3	2:	6	2	3	0	3	0
	3	0	1	1	3:	0	3	0	1	0	1
Col Sum		4	4		4:	1					
4	0	1	0	1	0:	1	0	1	0	1	0
	1	0	4	4	1:	0	1	0	4	0	4
	2	6	0	6	2:	10	2	6	0	6	0
	3	0	4	4	3:	0	3	0	4	0	4
	4	1	0	1	4:	5	4	1	0	1	0
Col Sum		8	8		5:	0					

Table 3.1: Weight distributions and patterns for 1-bit XOR checksums.

3. Since XOR produces only even weighed code words, the non-zero entries in the matrices follow a zig-zag pattern, which extends for wider checksums and more data words.

4. As shown in the "Position Pattern" matrices, there are two more abstract, alternating patterns: trapezia, and triangles. The patterns of the first two distributions have zero-length sides so that they appear as a line and a triangle. For wider checksums, the patterns extend to the other rows and columns. The upper right corner is always zero, while the lower left and lower right ones are alternatingly zero. Further, the entries in the matrix are *symmetric*: (a) The trapezia are point-symmetric, while (b) the triangles are line-symmetrical , as indicated by the dotted lines.

5. The *cross totals* of the downward-right (south east) diagonals, as highlighted in the "Next Rightmost" matrices, are the rightmost entries, in order, in the next matrix with one more data word.

6. The row sums of each matrix equal a row from the Pascal Triangle (the binomial coefficients). This is, because the data bits are distributed in that manner.

7. The column sums of each matrix are a power of two, multiplied by binomial coefficients:

$$\text{col\_sum}([\![\mathbb{X}]\!]) = \sum_{i=0}^{\#\mathbb{D}\cdot|\mathbb{X}|} \mathfrak{w}_{i,[\![\mathbb{X}]\!]}^{\#\mathbb{D}} = \binom{|\mathbb{X}|}{[\![\mathbb{X}]\!]} \cdot 2^{|\mathbb{X}|\cdot(\#\mathbb{D}-1)} \tag{3.40}$$

8. For each column $x = [\![\mathbb{X}]\!]$, all weights are divisible without remainder by the one element in the same column from the first matrix $\mathfrak{w}_{x,x}^{\#\mathbb{D}=1}$

3$^{\text{rd}}$-last weight 3$^{\text{rd}}$ weight

	b																		
$\#\mathbb{D}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	1	0	2	0	1														
2	1	0	6	0	9	0	0												
3	1	0	12	0	38	0	12	0	1										
4	1	0	20	0	110	0	100	0	25	0	0								
5	1	0	30	0	255	0	452	0	255	0	30	0	1						
6	1	0	42	0	511	0	1484	0	1519	0	490	0	49	0	0				
7	1	0	56	0	924	0	3976	0	6470	0	3976	0	924	0	56	0	1		
8	1	0	72	0	1548	0	9240	0	21942	0	21816	0	9324	0	1512	0	81	0	0

Table 3.2: Weight distribution triangle for 2-bit checksums. Each cell represents a $\kappa_b^{\text{XOR},|\mathbb{X}|=2,\#\mathbb{D}}$.

Weight Distribution Triangle

When plotting the determined weight distributions of a given checksum width with increasing numbers of data words, we obtain a *Weight Distribution Triangle*. For that, we can also infer some generic rules that can be derived directly from the weight distributions themselves. This resembles the incremental construction pattern of the Pascal Triangle. In Table 3.2 we exemplarily show the weight distributions for 2-bit XOR checksums and up to 8 data words. For all checksum widths greater than 1, the weight distributions can not be computed as easily as for the 1-bit case. However, from the counted distributions, we found the following integer sequences that seem to appear for *all* XOR checksum widths $[\![\mathbb{X}]\!]$. Since the weight distribution elements depend on the three parameters (1) bit flip weight b, (2) checksum width $|\mathbb{X}|$, and (3) number of code words $\#\mathbb{D}$, we will use the following notation: $\kappa_b^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}}$.

Column-Wise:

1. Column 0 is always one, just as in the Pascal Triangle, given for the sake of completeness, since zero-bit flips are not of interest for SDC probabilities.

$$\kappa_{b=0}^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}} = 1 \tag{3.41}$$

2. Every second column, i.e. those with an odd weight ($b \equiv 1 \bmod 2$), is zero, because the XOR checksum generates even parities across all bit lines.

3. The third column $b = 2$ (encircled in blue) equals the product of column 3 of the Pascal Triangle ($\#\mathbb{D} + 1$) and the checksum width $|\mathbb{X}|$:

$$\kappa_{b=2}^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}} = |\mathbb{X}| \cdot \binom{\#\mathbb{D}+1}{2}. \tag{3.42}$$

4. The last column is alternatingly one and zero, where for odd b it is one and for even b it is zero. This means that code word inversions (including the checksum) can only be detected when the number of covered data words $\#\mathbb{D}$ is *even*.

$$\kappa_{b=|\mathbb{X}|\cdot(\#\mathbb{D}+1)}^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}} = (\#\mathbb{D} \bmod 2) \tag{3.43}$$

5. For *odd* numbers of data words, *all* of the last $|\mathbb{X}|$ columns are zero.

Row-Wise:

1. For any checksum width $|\mathbb{X}|$, row $\#\mathbb{D} = 1$ equals the $|\mathbb{X}|$-th row from the Pascal Triangle, whereas a null is inserted in between each entry:

$$\kappa_b^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}=1} = \begin{cases} \binom{|\mathbb{X}|}{b/2} & b \equiv 0 \mod 2 \\ 0 & b \equiv 1 \mod 2 \end{cases} \tag{3.44}$$

2. The weight distribution elements of row $\#\mathbb{D} = 2$ can be directly computed as

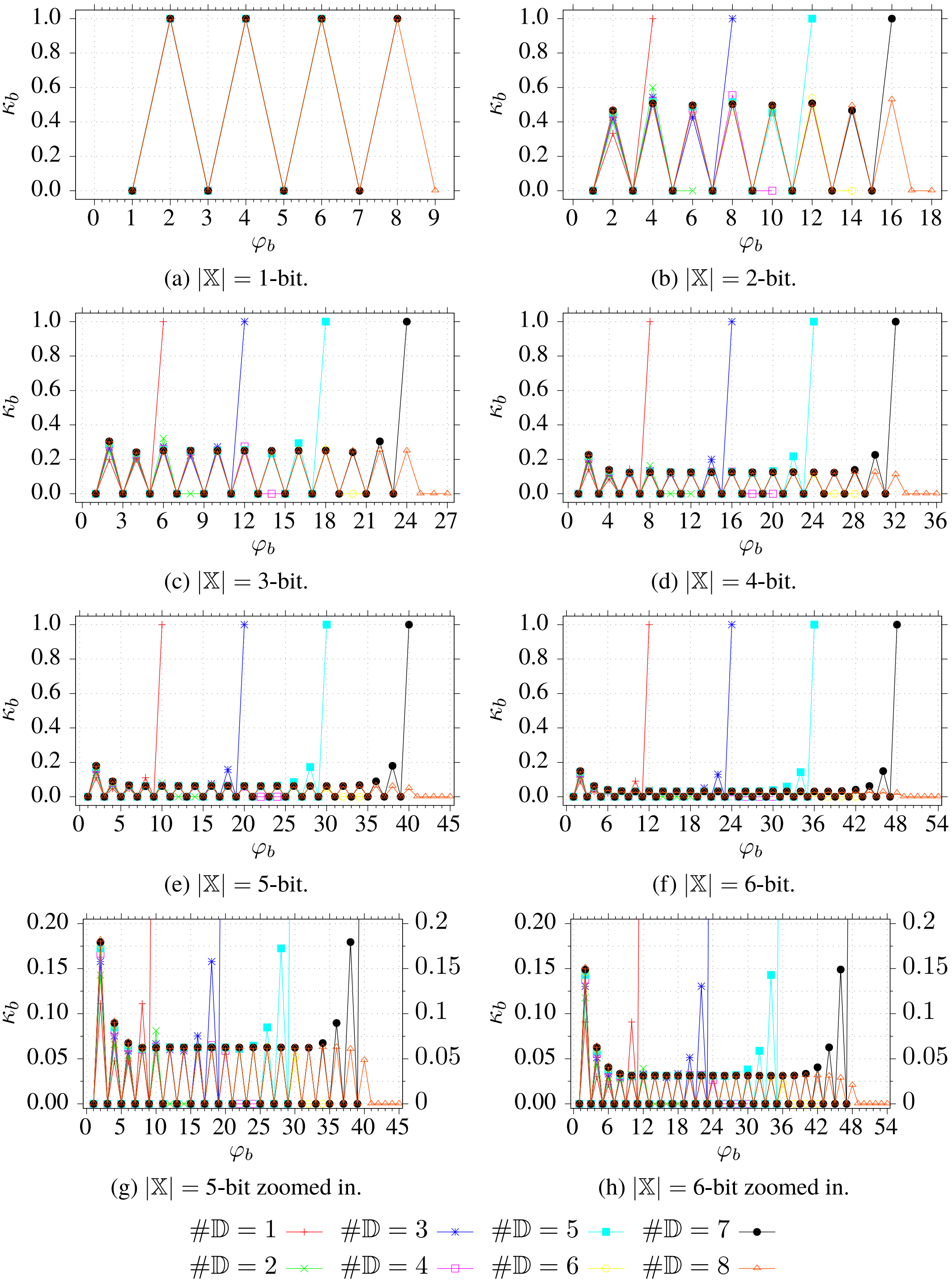
$$\kappa_b^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}=2} = \begin{cases} \kappa_b^{\text{XOR},|\mathbb{X}|,\#\mathbb{D}=1} \cdot 3^{b/2} & b \equiv 0 \mod 2 \\ 0 & b \equiv 1 \mod 2 \end{cases} \quad 0 \leq i \leq |\mathbb{C}| \tag{3.45}$$

The above findings can be found in all the distance distributions which we computed ($1 \leq |\mathbb{X}| \leq 6$ and $1 \leq \#\mathbb{D} \leq 8$). With the help of these findings, we can already iteratively compute all weight distributions for 1-, 2-, and 3-bit checksums from the weight distribution matrix $\mathbb{W}_{\text{XOR}}^{\#\mathbb{D},|\mathbb{X}|}$ alone. This is because the row sums in $\mathbb{W}_{\text{XOR}}^{\#\mathbb{D},|\mathbb{X}|}$ are known to be the appropriate binomial coefficients, because these reflect the data bit distribution. For the checksum weights up to 3, there are at most 4 columns in the matrix. Since we also know the two symmetry patterns and the patterns for constructing the numbers in the rightmost column, we can iteratively compute all the numbers in the weight distribution matrix. Summing up the elements of the upward-right (north east) diagonals delivers the actual weight distribution. For checksum codes with checksum width larger than 3, there are multiple unknown in several rows of the weight distribution matrix, for which we could not find deduction rules, yet.

Concrete Conditional Probabilities

We presented a first way towards determining the weight distribution for arbitrary checksum codes, i.e. for arbitrary checksum weights and arbitrary numbers of covered data words. 1-, 2-, and 3-bit checksums can be arbitrarily derived, but for wider checksums a brute force enumeration is still needed. Using Equation (3.28) we compute and display the conditional probabilities of SDC for XOR checksums with $1 \leq |\mathbb{X}| \leq 6$ and $1 \leq \#\mathbb{D} \leq 8$ in the following Figures 3.8a to 3.8f, respectively. For the 5- and 6-bit checksum cases, Figures 3.8g and 3.8h show a zoomed-in distribution, respectively. The figures reveal the following insights:

1. All distributions show a characteristic zig-zag pattern, where *all* odd-weight bit flips are detected.
2. For parity bits ($|\mathbb{X}| = 1$), the SDC probability of *any* even bit flip weight is exactly *one*.
3. For parity words,
   (a) when the number of data words $\#\mathbb{D}$ is *odd*, then code word *inversions* cannot be detected.
   (b) Furthermore, for *odd* $\#\mathbb{D}$, the SDC probability *increases* for the high bit flips weights, while for *even* $\#\mathbb{D}$ the probabilities first decrease and then drop to zero (cf. $\#\mathbb{D} \in \{7, 8\}$ in Figures 3.8g and 3.8h).



(a) $|\mathbb{X}| = 1$-bit.

(b) $|\mathbb{X}| = 2$-bit.

(c) $|\mathbb{X}| = 3$-bit.

(d) $|\mathbb{X}| = 4$-bit.

(e) $|\mathbb{X}| = 5$-bit.

(f) $|\mathbb{X}| = 6$-bit.

(g) $|\mathbb{X}| = 5$-bit zoomed in.

(h) $|\mathbb{X}| = 6$-bit zoomed in.

Figure 3.8: Conditional SDC probabilities for 1- to 6-bit XOR checksums.

(c) Increasing numbers of data words have a negative impact on the detection rate for the low end of bit flip weights. With each additional data word, the SDC probability increases, especially for double-bit flips and then heavily decreases again (cf. Figures 3.8g and 3.8h).

4. For *even* numbers of data words $\#\mathbb{D}$, the last $|\mathbb{X}|$ bit flip weights are *zero*.

5. Finally, in general the wider the *checksum*, the lower the probabilities for the middle bit flip weights. For a fixed checksum width, the curves for increasing numbers of data words $\#\mathbb{D}$ are pretty close to each other in the middle part, except towards the lower and upper bound of bit flip weights b.

3.2.3 Probabilities of AN Codes

Obtaining the SDC probabilities of AN codes is in general much harder than for codes like Hamming or XOR. This is because the latter are linear codes where it suffices to count all the code word weights. AN codes, and arithmetic codes in general, are *non-linear* codes so that we have to count *all* code word distances, i.e. exhaustively compare all code words and compute the Hamming distances (d_H) between each and every code word. Furthermore, due to the convolution of the multiplication used for hardening, each instance $A \in \mathbb{A}$ has different error detection properties with regard to the data type width $|\mathbb{D}_\Theta|$. Earlier work by Massey [113] only considers the *arithmetic* weights and distances. These do not apply here. We need the Hamming distances and weights, because we do not assume arithmetic errors, but bit flips. He and previous work use the term *linear residue codes*, where *arithmetic* linearity is meant and not the linearity as for Hamming and XOR checksum codes. To avoid confusion we will refrain from using the notion of linear residue codes. For AN codes, Hoffmann *et al.* computed the probabilities for data widths of 8 and 16 bits and As up to 8 and 16 bits, respectively. They exhaustively computed all Hamming distances between all code words, but only consider 2 code widths of 16 and 32 bits. However, the results from the previous work are insufficient for the database domain, because

1. possibly all data bit widths between 1 and 32 bits are to be supported [189], and

2. larger As may be required for future error models.

Enumerating the Distance Distributions

The challenge is to obtain the distance distribution counts $\kappa_b^{\mathbb{C}}$ in an efficient manner. For AN coding, the convolution of the multiplication (hardening) cannot be described in a way which allows simplifications. This part was developed together with Matthias Werner, a colleague from the Center for Information Services and High Performance Computing at the Technische Universität Dresden and this work was published in [184, 185]. Matthias Werner proposed the idea to try the grid approximation, implemented the Cuda code for graphics processing units (GPUs), and was responsible for submitting the compute jobs to the Taurus Bull cluster. We use the following function to describe the naive approach:

$$\delta_b(x,y) = \begin{cases} 1, & \text{if } d_\mathrm{H}(x,y) = b, \\ 0, & \text{if } d_\mathrm{H}(x,y) \neq b, \end{cases} \qquad 0 \leq b \leq n\,.$$

Then, we can compute the weight distribution as

$$\kappa_b^{\mathbb{C}} = \sum_{\alpha \in \mathbb{C}} \sum_{\beta \in \mathbb{C}} \delta(\alpha, \beta)\,. \tag{3.46}$$

The complexity of Equation (3.46) is $\mathcal{O}(4^k)$. This is because we have to inspect all possible *pairs* of code words, so that the number of distances $\#d_H$ which must be enumerated is

$$\#d_H = \binom{2^{|\mathbb{D}|}}{2}. \tag{3.47}$$

For an additional data bit, we have twice as many code words, so that it increases to

$$\#d_H = \binom{2^{|\mathbb{D}|+1}}{2} = \binom{2 \cdot 2^{|\mathbb{D}|}}{2}. \tag{3.48}$$

As a consequence, the effort about *quadruples* by every additional data word bit.

Proof. To show that Equation (3.47) develops to a quadruped increase, we write out in full the binomial coefficient, first with generic parameter x and we set $2^{|\mathbb{D}|} = y$:

$$\begin{aligned}
\binom{2 \cdot 2^{|\mathbb{D}|}}{x} =& \binom{2y}{x} \\
=& \frac{(2y)!}{x! \cdot (2y-x)!} \\
=& \frac{1 \cdot 2 \cdot \ldots \cdot y \cdot (y+1) \cdot \ldots \cdot (2y-x) \cdot (2y-x+1) \cdot \ldots \cdot 2y}{x! \cdot 1 \cdot 2 \cdot \ldots \cdot y \cdot (y+1) \cdot \ldots \cdot (2y-x)} \\
=& \frac{(y-x+1) \cdot (y-x+2) \cdot \ldots \cdot y}{x!} \\
& \cdot \frac{(y+1) \cdot \ldots \cdot (2y-x) \cdot (2y-x+1) \cdot \ldots \cdot 2y}{(y-x+1) \cdot (y-x+2) \cdot \ldots \cdot y \cdot (y+1) \cdot \ldots \cdot (2y-x)} \\
=& \binom{y}{x} \cdot \frac{(2y-x+1) \cdot \ldots \cdot 2y}{(y-x+1) \cdot \ldots \cdot y}
\end{aligned}$$

Next, we set $x = 2$ as in Equation (3.47):

$$\begin{aligned}
\binom{2y}{2} &= \binom{y}{2} \cdot \frac{(2y-2+1) \cdot (2y-2+2) \cdot \ldots \cdot 2y}{(y-2+1) \cdot (y-2+2) \cdot \ldots \cdot y} \\
&= \binom{y}{2} \cdot \frac{(2y-1) \cdot 2y}{(y-1) \cdot y} \\
&\approx \binom{y}{2} \cdot \frac{2y \cdot 2y}{y \cdot y} \\
&= 4\binom{y}{2}.
\end{aligned}$$

□

$\|\mathbb{D}\| = n$	1	2	3	4	5	6	7	8
$\#\mathbb{D}$	2	4	8	16	32	64	128	256
combinations $\#d_{H,n}$	1	6	28	120	496	2016	8128	32640
relative increase $\frac{\#d_{H,n}}{\#d_{H,n-1}}$	–	6	$4.\overline{6}$	4.29	$4.1\overline{3}$	4.06	4.03	4.016

Table 3.3: Concrete numbers for the relative increase after Equation (3.48).

Instances for the number of combinations and the relative increase are given in Table 3.3. In practice, the number of combinations only half as much due to the symmetry: between two code words, only one edge is computed and can then simply be counted twice. But still, since this is an exponential increase with a factor of 4, increasing data widths make it incredibly hard to determine SDC probabilities using the exhaustive approach. For instance, relative to the effort to compute the probabilities for a single A for 32-bit data, the effort to compute the probability for a single A for 64-bit data is $4^{64-32} = 4^{32} \approx 1.8 \cdot 10^{19}\times$ as high. For parameter optimization, this must be run many thousands of times, depending on the width of A. Especially, for AN coding each odd A must be examined again for each data width $|\mathbb{D}_\Theta|$. We call this naive approach *exact*, as it examines all code words. For AN coding, Table 3.4 shows runtimes for the exact computation of the weight distribution for a single A using a single CPU, a single GPU, or a small cluster of 4 GPUs on the left half. While for a single A the measured runtimes are not a problem, be reminded that all odd As must be tested for a given parameter width. Using the numbers from Table 3.4, we can extrapolate the necessary runtimes for 32-bit data with the help of Equation (3.48), whereas for simplicity we use a factor of 4. The results are visualized in Figure 3.9. For the exact approach, using a single CPU would take about 47.6 years, while using a single GPU would still take about 12.3 years, and finally using the small 4-GPU cluster would take 3.4 years.

To *mitigate* the huge runtimes, we use a sampling-based approach, which approximates the weight distribution by comparing only a subset of all code words. Here, the main problem is the *distribution* function for choosing the subset of code words. We investigated three different distributions: pseudo-random (σ_{pseudo}), quasi-random (σ_{quasi}), and grid-point (σ_{grid}). Note that σ_{pseudo} is prone to clustering, while σ_{quasi} fills the space more uniformly. The probabilistic error of Monte-Carlo (pseudo-random) is known to be $\mathcal{O}(1/\sqrt{M})$ and for quasi-Monte-Carlo it is $\mathcal{O}((\log M)^q/M)$ with number of dimensions q and number of iterations M [103]. The grid-point approach chooses regularly aligned samples, given by $\sigma_{\text{grid}}(r) = (2^k \cdot r)/M$. If $M = 2^k$, then the grid sampling yields the correct result, while random numbers still miss the solution due to collisions and gaps.

	exact		
k	t_{CPU}	$t_{1\cdot\text{GPU}}$	$t_{4\cdot\text{GPU}}$
8	7 ms	1 ms	3 ms
16	376 ms	130 ms	41 ms
24	382 min	99 min	27 min
32	–	–	–

Table 3.4: Runtimes for computing the distance distributions of AN codes for $A = 61$. Average values after 5 runs on the Bull HPC-Cluster Taurus at TU Dresden. CPU: 2×E5-2680 v3 Haswell 12-core 2.50 GHz, gcc5.3, OpenMP 4.0. GPU: NVIDIA Tesla K80, CUDA 7.5

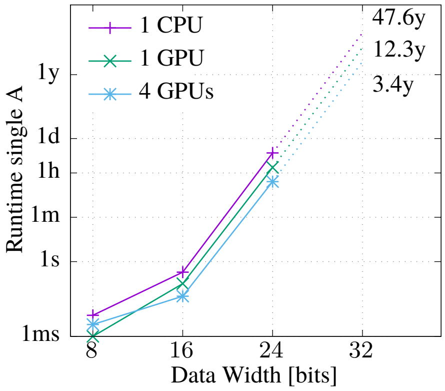


Figure 3.9: Extrapolation of the runtimes from Table 3.4

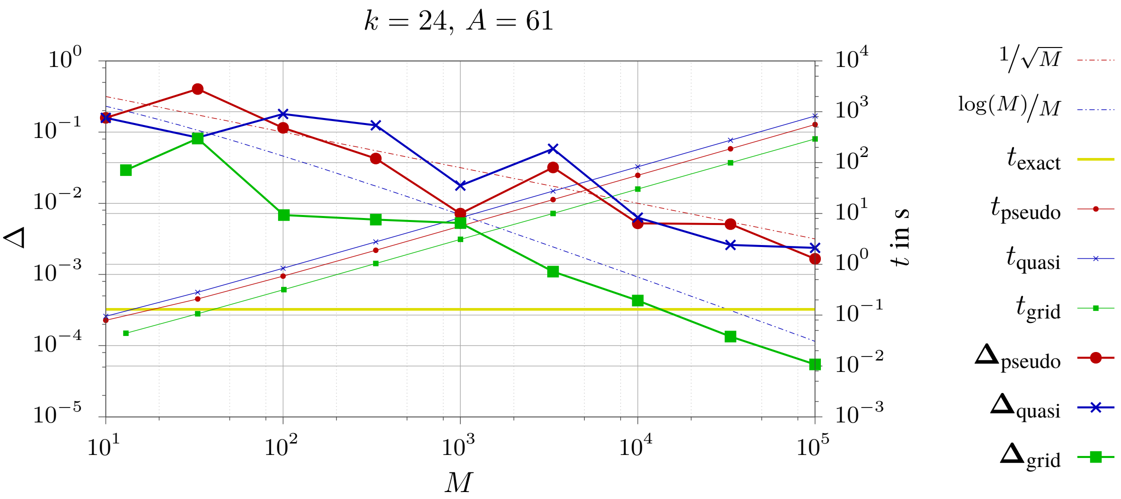


Figure 3.10: Convergence of maximum relative error Δ and runtime t according to the number of iterations M.

Algorithm 3.1 AN code distance distribution – basic algorithm

```
Input: $k \geq 2$ ($k = |\mathbb{D}|$)
Input: Value $A > 0$, $n = k + h$, $h = \lceil \log_2(A) \rceil$
Input: Initial distance distribution $c_b^A = 0$, $b = 0, \ldots, n$
Output: Distance distribution $c^A$ of code $C_A$
1: for $\alpha = 0, \ldots, (2^k - 1)$ do                ▷ outer loop is parallelized on GPU[s]
2:     for $\beta = \sigma_{\text{grid}}(r),\ r = 0, \ldots, M$ do    ▷ inner loop is processed by each thread
3:         $b \leftarrow d_{\text{H}}(A\alpha, A\beta)$
4:         $\kappa_b^A \leftarrow \kappa_b^A + 1$
5:     end for
6: end for
7: return $K^A = \{\kappa_b^A\}$
```

Figure 3.10 shows a comparison between the three distributions of convergence and runtime for the case $k = 24$ and $A = 61 \Rightarrow n = 30$ and includes the theoretic Monte-Carlo error boundaries. Pseudo- and quasi-random numbers were generated with the cuRAND library. The 1D grid approximation outperforms the random distributions in virtually all cases, yielding smaller error Δ and lower runtime t. It is, furthermore, directly influenced by the value of M, and we found that *odd* values lead to much smaller errors than even ones.

Algorithm 3.1 shows the 1D grid approach for enumerating the weight distribution of an AN code. For GPU clusters, we distribute the outer loop evenly across the GPUs. When symmetry is exploited in line 2, the workload size of each GPU is computed by:

$$\lceil 2^k \omega_{i+1} \rceil - \lceil 2^k \omega_i \rceil,\ w_i = 1 - \sqrt{1 - {}^{i}\!/\!{}_{N}} \quad , 0 \leq i < N\text{=\#GPUs} \tag{3.49}$$

ω_i is the solution of $\displaystyle\int_i^{i+1} 1 - x \,\mathrm{d}x = {}^{1}\!/\!{}_{N}$ for equal work size areas. The maximal relative error of the estimation $\hat{c}_b^A$ is given by

$$\Delta = \max_{b>0} \frac{\mid \kappa_b^A - \hat{\kappa}_b^A \mid}{\kappa_b^A},$$

where $b = 0$ is omitted due to $\kappa_0^A = 2^k$. Algorithm 3.1 can be parallelized on GPUs, since the Hamming distances of two code words can be computed independently. We use CUDA C/C++ for programming Nvidia GPUs[3]. As registers of GPUs are 32-bit wide, the multi-GPU implementation uses 32-bit integers as long as the array elements in a thread do not overflow. From Equation (3.28) follows

$$\max_b \kappa_b^A \leq \max_b 2^k \binom{n}{b} = 2^k \binom{n}{n/2}$$

and the upper bound for using 32-bit integers is:

$$\kappa_{b,\mathrm{thread}}^A \leq \frac{2^k \binom{n}{n/2}}{\mathrm{threads}} < 2^{32} \;.$$

As the GPU uses 64 bits for the global array, the highest data word width for the GPU algorithm is $k = 33$. There are many more technical details to tailor the counting towards the wide variety of code widths and the details are presented in [184, 185].

We used both the exact and grid-point approaches on the multi-GPU cluster to exhaustively enumerate the distance distributions for *all* data widths $1 \leq |\mathbb{D}| \leq 32$ and all widths of A $1 \leq A \leq 16$[4]. This goes further than in any previous work, since many more data widths are taken into account. In Figure 3.11, we show examples of the resulting conditional SDC probabilities for data widths 8 and 16 and $A \in \{17, 19, 21, \ldots, 127\}$. The As are divided into three classes according to their bit width, i.e. $|A| \in \{5, 6, 7\}$. Additionally, Figure 3.12 provides more examples for $k = 24$ and $|A| \in \{5, 10, 16\}$. We observe the following:

1. Increasing $|A|$ generally lead to decreased SDC probabilities, at the expense of larger code words.
2. With increasing values of A (the parameter itself), the resulting pattern looks like a seesaw. Generally speaking, for As between two consecutive powers of 2, larger values of A lead to *more balanced* SDC probabilities.
3. Especially for low and high b, some As exhibit anomalies in the form of very high (bad) probabilities, which confirms previous findings [69, 161]. This means that these parameters are very bad for detecting bit flip weights equal to the minimal Hamming distance, or equal or close to the code word width.
4. The "clutter" of the lines, which is very prominent for $k = |\mathbb{D}| = 8$, is reduced with increasing code bit widths $|\mathbb{C}|$. For instance, The lines for $k = |\mathbb{D}| = 16$ and $|A| = 7$ are much smoother than all the other lines, of course except for the outliers at the front and end.

Figure 3.12a confirms how increasing $|A|$ reduce the probabilities for undetected bit flips and it shows difference of shapes between the lower and upper ends per $|A|$. Figure 3.12b shows how the different As generate codes with much variety for the low and high b-borders. The flat regions

[3] The sources are available on `https://github.com/brics-db/coding_reliability/`

[4] The raw data and scripts are available on `https://brics-db.github.io`

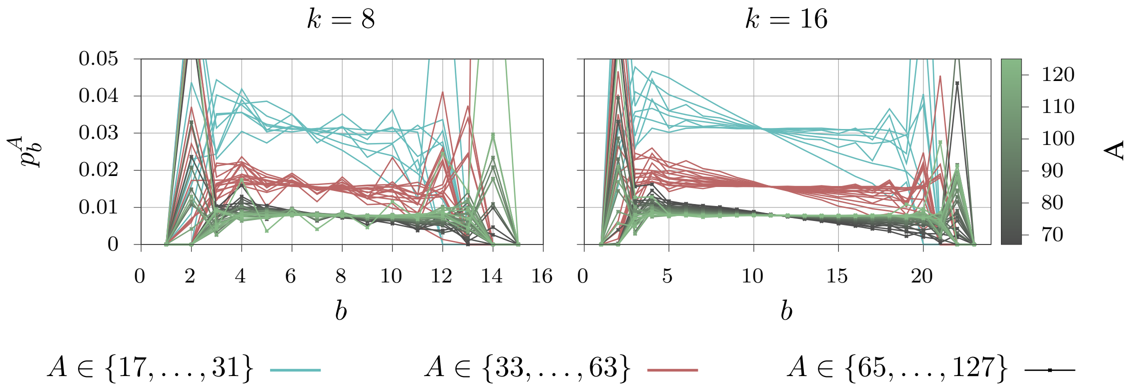
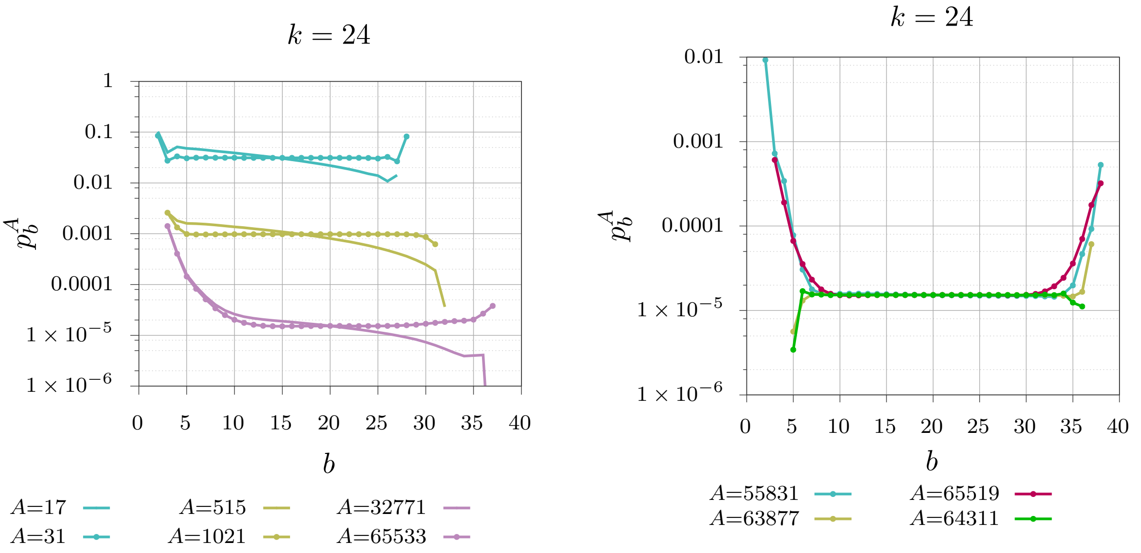


Figure 3.11: How parameter A influences the conditional SDC probability $\varphi_b^{\mathbb{C}}$, for $k = 8$ and $k = 16$. The $A \in \{65, \dots, 127\}$ are additionally color coded from black to dark green.



(a) As close to powers of 2 with $|A| \in \{5, 10, 16\}$

(b) More cases for $|A| = 16$ showing great differences at the two ends.

Figure 3.12: How parameter A influences the conditional SDC probability $\varphi_b^{\mathbb{C}}$, for $k = 24$.

	exact			$\sigma_{\text{grid,1D,4·GPU}}$		
k	t_{CPU}	$t_{1\cdot\text{GPU}}$	$t_{4\cdot\text{GPU}}$	t_M	Δ_M	M
8	7 ms	1 ms	3 ms	6 ms	0.0232	101
16	376 ms	130 ms	41 ms	11 ms	0.0031	1001
24	382 min	99 min	27 min	354 ms	0.0053	1001
32	–	–	–	5 min	–	1001

Table 3.5: Runtimes for exact and grid approximation methods, extending Table 3.4.

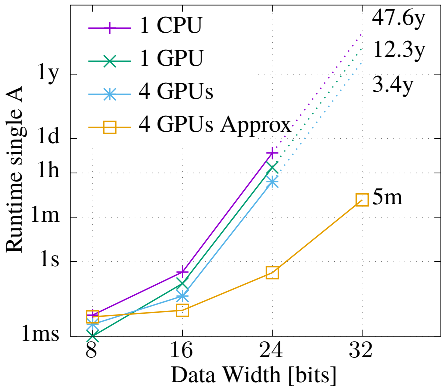


Figure 3.13: Visual comparison of the runtimes from Table 3.5

follow the 2^{-h}, $h = \lceil \log_2 A \rceil$ bound. A perfectly balanced code would follow this bound across all weights, but this is only an utopia like unicorns. The patterns show that As which are *arithmetically* close together may result in great variety of their detection capabilities.

Finally, we compare the runtimes between the exact and grid approximation methods in Table 3.5 and fig. 3.13. We can see how the grid approximation practically allows to compute the SDC probabilities for 32-bit data and beyond. For a single A on 32-bit data using the 4-GPU cluster, this only takes 5 minutes, which is more than *five orders of magnitude* faster than the exact method.

Optimality Criteria

With the help of the distance distributions we can now choose desired As. Therefore, we have to define an optimality criterion. Like previous work [69, 161], we start with a *set* of possible "super A"s:

Definition 9 (Super As). *For a given parameter bit width $h = |A|$ and data bit width $k = |\mathbb{D}|$, the* set *of "super A"s denoted as $A^{\sigma}_{h,k} \in \mathbb{A}^{\sigma}_{h,k}$ contains those A which generate AN codes of width $n = h + k$ with* largest *minimal Hamming distance $d_{\text{H,min}}$.*

This greatly reduces the set of possible parameter candidates. However, this generates almost always sets with multiple candidates. We now further restrict this set of super As to single candidates:

Definition 10 (Golden A). *For a given parameter bit width $h = |A|$ and data bit width $k = |\mathbb{D}|$, a "golden A" $A^{\Psi}_{h,k}$ is a super A which satisfies some optimality criterion Ψ.*

There are multiple possible choices for Ψ, which are *application specific* and may depend on what error characteristics some specific hardware may exhibit. Consequently, the objective function chosen for a concrete optimality criterion will most probably be tailored towards a specific hardware error model. This, in turn, could be delivered by the hardware vendor for compile-time decisions, or by the hardware itself at run-time for dynamic adaptation. However, this is beyond the scope of this thesis and we will return to that subject in the future work Section 7.1. Since there are no concrete hardware error models yet available, we will first define two very generic objective functions. The first objective function

$$\eta(k, A) = \max b \quad , \varphi_x^{k,A} = 0 \wedge 0 \leq x \leq b \tag{3.50}$$

returns the *minimal bit flip weight* that is detectable by the AN code defined by the given A and data bit width k. The second objective function

$$\psi(k, A) = \varphi_{b=\eta(k,A)+1}^{k,A} \tag{3.51}$$

selects the first non-zero conditional SDC probability for the AN code defined by the given A and data bit width k. Using these two objective functions, we define our first optimality criterion

$$\Psi_\eta(k, h) = \max\big(\eta(k, A), 1 - \psi(k, A)\big) \quad , \ |A| = h, \tag{3.52}$$

which selects for a given data bit width k and parameter bit width h the one A which has the largest minimal detectable bit flip weight and the lowest first non-zero conditional SDC probability.

Applying Ψ_η to the various combinations $1 \leq k \leq 32$ and $2 \leq h \leq 16$, we obtain 480 golden As, listed in the appendix in Tables A.1 to A.4. While this is the complete list for Ψ_η, in a real database system we may be only interested in the *smallest A* for a given minimal detectable bit flip weight η and some data bit width k. The result is shown in Table 3.6, which may serve as a static selector for parameter A, where a real database system may choose at run-time appropriate AN codes depending on data bit widths or other data characteristics. Since we only computed A up to 16 bits, up to now, the table contains empty cells.

We noted above that the optimality criterion is application specific. In other contexts, other criteria may be required, e.g. choosing those super As which minimize the area underneath the conditional SDC probability distribution curve. The third objective function

$$A_\varphi(k, A) = \sum_{i=2}^{n} \left(\frac{\max(\varphi_{i-1},\, \varphi_i) - \min(\varphi_{i-1},\, \varphi_i)}{2} + \min(\varphi_{i-1},\, \varphi_i) \right) \quad , \ n = |\mathbb{C}|, \ \varphi_x \equiv \varphi_x^{k,A} \tag{3.53}$$

computes the area underneath the SDC probability distribution curve for a code $\mathbb{C}$. An appropriate optimality criterion is then defined as

$$\Psi_{A_\varphi}(k, h) = \min\big(A_\varphi(k, A)\big) \quad , \ |A| = h, \tag{3.54}$$

which selects for a given data bit width k and parameter bit width h the one A which has the smallest area under its SDC probability curve, *irrespective* of the minimal detectable bit flip weight η. Using Equation (3.54) results in two differences. First, for virtually all categories (k, h) the As differ. Second, for the same combination (k, h) we sometimes select golden As which have smaller η than in Table 3.6. To circumvent the latter, we can combine the two previous optimality criteria into the third one, given by

$$\Psi_{\eta,A_\varphi}(k, h) = \min\big(n - \eta(k, A), A_\varphi(k, A)\big) \quad , \ |A| = h, \ n = k + h = |\mathbb{C}|, \tag{3.55}$$

$\lvert\mathbb{D}_\Theta\rvert$	η – minimal detectable bit flip weight							
	1	2	3	4	5	6	7	8
1	**3**/2	**7**/3	15/4	**31**/5	63/6	**127**/7	255/8	511/9
2	**3**/2	**13**/4	**53**/6	213/8	**853**/10	3285/12	13141/14	52565/16
3	**3**/2	**29**/5	45/6	**467**/9	1837/11	**7349**/13	23733/15	
4	**3**/2	27/5	**89**/7	933/10	6777/13	31385/15		
5	**3**/2	**29**/5	117/7	933/10	7085/13	31373/15		
6	**3**/2	**29**/5	**233**/8	1899/11	7837/13	62739/16		
7	**3**/2	**29**/5	217/8	1803/11	**13963**/14	55831/16		
8	**3**/2	**29**/5	**233**/8	1939/11	**13963**/14	55831/16		
9	**3**/2	**29**/5	185/8	1939/11	15717/14	55831/16		
10	**3**/2	**61**/6	185/8	**3739**/12	27425/15			
11	**3**/2	**61**/6	451/9	**3739**/12	27425/15			
12	**3**/2	**61**/6	**463**/9	3737/12	29925/15			
13	**3**/2	**61**/6	**463**/9	3349/12	27825/15			
14	**3**/2	**61**/6	**463**/9	6717/13	63877/16			
15	**3**/2	**61**/6	**463**/9	7785/13	63877/16			
16	**3**/2	**61**/6	**463**/9	7785/13	63877/16			
17	**3**/2	**61**/6	393/9	7785/13	63859/16			
18	**3**/2	**61**/6	**947**/10	7785/13	63859/16			
19	**3**/2	**61**/6	**947**/10	7985/13				
20	**3**/2	**61**/6	985/10	7985/13				
21	**3**/2	**61**/6	985/10	15507/14				
22	**3**/2	**61**/6	985/10	15993/14				
23	**3**/2	**61**/6	985/10	15993/14				
24	**3**/2	**61**/6	981/10	15993/14				
25	**3**/2	111/7	981/10	15685/14				
26	**3**/2	111/7	981/10	15203/14				
27	**3**/2	111/7	951/10	15203/14				
28	**3**/2	111/7	951/10	29685/15				
29	**3**/2	111/7	835/10	*29685/15				
30	**3**/2	125/7	835/10	*31693/15				
31	**3**/2	125/7	**881**/10	*32211/15				
32	**3**/2	125/7	**881**/10	*32417/15				

Table 3.6: Golden As per minimal detectable bit flip weight η after $\Psi_\eta(k,h)$ – Equation (3.52) – in the form $A/|A|$. **Bold** numbers are prime. *-numbers are obtained through grid approximation.

which selects for a given data bit width k and parameter bit width h the one A which has the largest minimal detectable bit flip weight and the smallest area under its SDC probability curve. The golden As obtained here differ from the second optimality criterion only in those situations where the minimal detectable bit flip weight η was smaller than for the first criterion. This means, all golden As selected by Equation (3.55) have the same η as those obtained by the first optimality criterion from Equation (3.52).

3.2.4 Concrete Error Models

So far we argued that there are no concrete hardware error models available for future hardware. We will now have a look at two transmission channel error models, which are considered in the literature: (1) equal probability of all bit flip weights and patterns [53, 69], and (2) the binary symmetric channel.

Equal Probabilities

In the first model, all bit flip weights and bit flip patterns are equally probable. When this assumption holds, then each bit flip weight $b \in 1, \ldots, n$ with $n = |\mathbb{C}|$ occurs with the same probability

$$P_b = {}^1\!/\!{}_n,$$

because there are n code bits. Consequently, the unconditional SDC probabilities of a code are equal to the conditional ones, except for that constant factor ${}^1\!/\!{}_n$. This factor is required, so that all unconditional probabilities for detectable and undetectable bit flips sum up to 1 and not to n. In this model, there is a bound p_{res} on the error detection capability called *residual error probability* [53, 69], which is computed using Equations (3.25) and (3.27) as

$$\begin{aligned} p_{\text{res}} &= \frac{\text{undetectable errors}}{\text{total errors}} = \frac{\sum_{i=0}^{t}\binom{n}{i}\cdot(2^k-1)\cdot 2^k}{\sum_{b=1}^{n}\beta_b^{\mathbb{C}}} = \frac{(2^k-1)\cdot 2^k}{2^k\cdot\sum_{b=1}^{n}\binom{n}{b}}\cdot\sum_{i=0}^{t}\binom{n}{i} \\ &= \frac{2^k-1}{2^n-1}\cdot\sum_{i=0}^{t}\binom{n}{i} \approx \frac{2^k}{2^n}\cdot\sum_{i=0}^{t}\binom{n}{i} = \underline{\underline{2^{-h}\cdot\sum_{i=0}^{t}\binom{n}{i}}}, \end{aligned} \tag{3.56}$$

with $n-k=h$. We will discuss the correlation between the conditional error probabilities and the residual error probability later in the comparison in Section 3.4.1. For detecting codes, we again set $t=0$, which results in

$$p_{\text{res,t=0}} = 2^{-h},$$

which turns out to be the same as in [53, 69]. For an SECDED code like Hamming, we set $t=1$, which results in

$$p_{\text{res,t=1}} = (1+n)\cdot 2^{-h}.$$

The residual error probability can then also be adapted to the error model using ${}^1\!/\!{}_n$. The unconditional probability for this error model, $P(SDC)_{\text{Eq}}^{\mathbb{C}}$, for a bit flip weight b is computed as

$$P(SDC)_{\text{Eq}}^{\mathbb{C}} = \sum_{b=1}^{n=|\mathbb{C}|}\left(P(\text{SDC}|b)^{\mathbb{C}}\cdot P_b\right) = {}^1\!/\!{}_n\sum_{b=1}^{n=|\mathbb{C}|}\varphi_b^{\mathbb{C}} \tag{3.57}$$

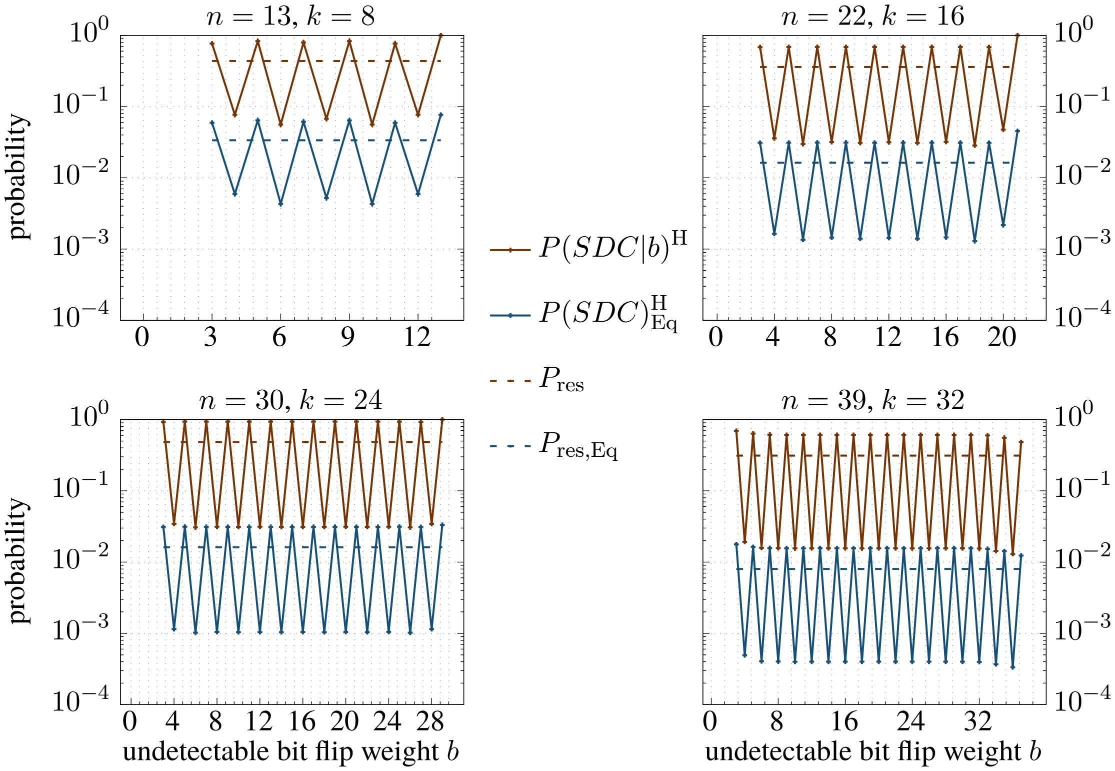


Figure 3.14: Unconditional and conditional SDC probabilities under the equal probabilities error model for Hamming

Examples for the conditional SDC probabilities under the equal probabilities error model are shown in Figure 3.14. There we see the distributions for four different data widths $|\mathbb{D}| \in \{8, 16, 24, 32\}$ for the respective shortened correcting Extended Hamming Codes. It shows how the error model decreases the SDC probabilities due to the assumption of the equal probabilities of patterns and bit flip widths, i.e. due to the factor $1/n$. The residual error probabilities are also plotted and this shows that the correction capability increases the SDC probabilities for some bit flip weights above this bound. The distributions of AN and XOR codes are similarly shifted downwards to lower probabilities.

Binary symmetric channel

One other widely used error model in the coding theory domain is the binary symmetric channel (BSC), depicted in Figure 3.15. There, the probability that a bit is flipped during the transmission

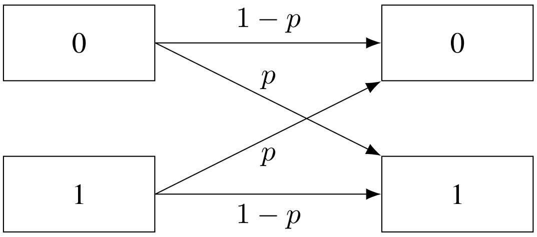


Figure 3.15: The binary symmetric channel (BSC) error model.

is p, while the chances of receiving a bit undisturbed is $1 - p$. It is furthermore a common assumption that bit flips are *independent*, which leads to the chance P_c that a code word of size n bits is transmitted correctly as

$$P_c = (1 - p)^n. \tag{3.58}$$

The probability P_b that inside an n-bit code word b bits flip is

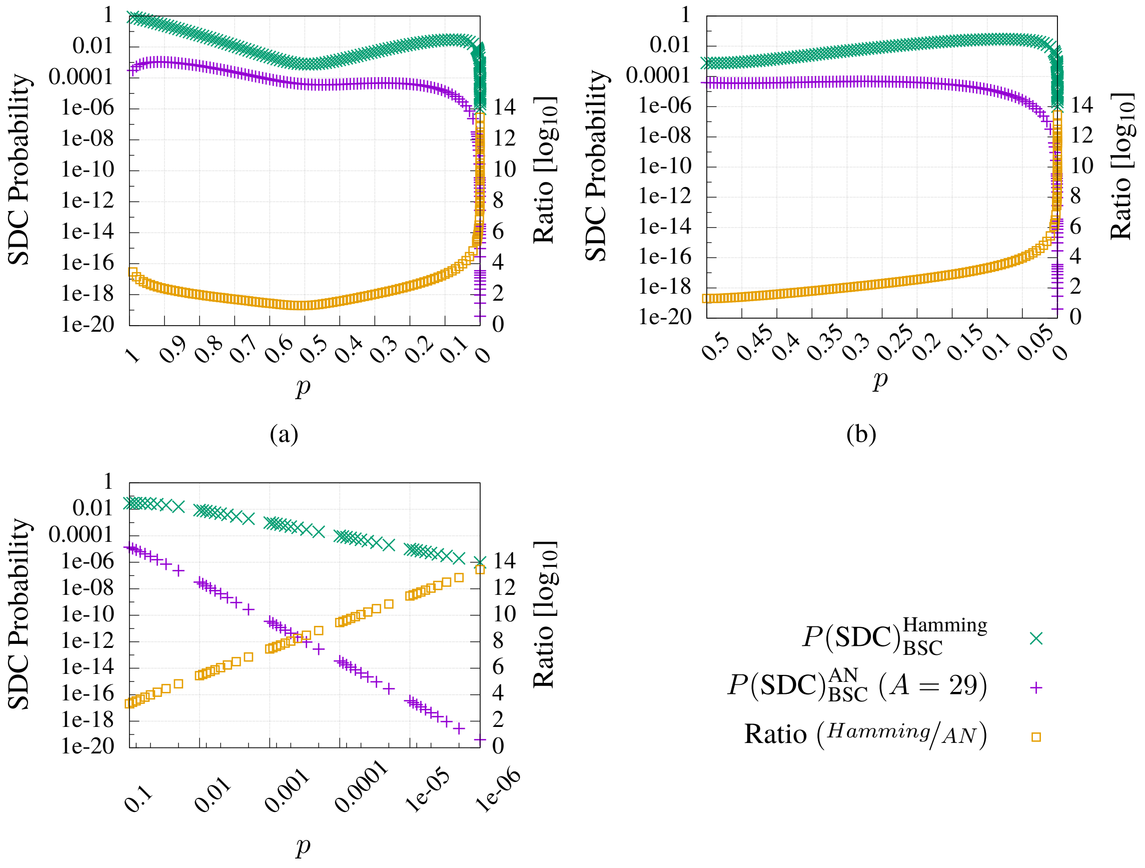
$$P_b = (1 - b)^{n-b} \cdot p^b. \tag{3.59}$$

The unconditional SDC probability for a BSC, a code $\mathbb{C}$, and some bit flip probability p is then obtained using Equation (3.28) by

$$P(SDC)^{\mathbb{C}}_{\text{BSC},p} = \sum_{b=1}^{n=|\mathbb{C}|} \left(P(\text{SDC}|b)^{\mathbb{C}} \cdot P_b\right) = \sum_{b=1}^{n=|\mathbb{C}|} \left(\varphi^{\mathbb{C}}_b \cdot (1 - b)^{n-b} \cdot p^b\right) \tag{3.60}$$

Figure 3.16 shows the effect of varying probability p when employing the BSC, i.e. Equation (3.60). There, we compare a (13,8,4) correcting shortened Extended Hamming code against a (13,8,3) AN code with the golden $A = 29$. By that, both codes have the same code width and the same minimal detectable bit flip weight η. On the x-axis we vary the bit flip probability p in various ranges to show different levels of detail. Figure 3.16a *highlights* $p \in [0.99, 0.1]$, where we see very different curves for the Hamming and AN code. First, for very high probability of flipping bits ($p \rightarrow 0.99$), Hamming has very high probability of SDC, whereas the AN code starts at a much lower total probability and first has a short *increasing* part for $p \in [0.99, 0.92]$. This shows that the AN code can detect large burst errors with much better probability. Interestingly, the total SDC probability decreases until $p = 0.48$ for Hamming and until $p = 0.45$ for AN and then rises again. This is, because then, lower numbers of bit flips become more probable where especially the Hamming code then performs worse and worse *in this interval*. The ratio between the two codes is given in *orders of magnitude* by $P^{\text{Hamming}}_{\text{BSC}}/P^{\text{AN}}_{\text{BSC}}$. In this interval, AN is up to 3.5 orders of magnitude better than the Hamming code with the same code width. Figure 3.16b reveals that in the lower range $p \in [0.5, 0.05]$, the SDC probability decreases greatly again for both codes, whereas for the AN code the decrease starts much earlier (at $p \approx 0.275$) than for Hamming (at $p \approx 0.08$). This means that the AN code is much more *balanced*, concerning the SDC probability, than the Hamming code. Zooming further in, Figure 3.16c shows that for even lower bit flip probabilities ($p < 0.1$), the AN code can far outperform the Hamming code. AN gets better and better compared to the Hamming code with decreasing p, being almost 13.5 orders of magnitude better than Hamming for $p = 10^{-6}$.

The above shows that the conditional SDC probability can be combined with concrete error models. By that, when concrete hardware error models become available, all of the conditional probabilities which we counted can be transformed into *unconditional* probabilities. Furthermore, new objective functions can be defined, which are tailored to these models. Based on that or other characteristics of the application, the environment, the hardware etc., new optimality criteria could be defined. Consequently, with all of the above findings we present a toolbox for choosing appropriate coding schemes based on desired data bit widths, redundancy margins, and error detection capabilities.



(a)

(b)

(c)

Figure 3.16: Concrete SDC probabilities for Hamming and AN (A=29) codes, with both k=8 and n=13.

3.2.5 Summary and Conclusions

No code is perfect in the sense that it could detect all bit flips. The quality of a code regarding effectiveness (Requirement $\mathcal{R}1$) is determined by the probability of silent data corruption. It means that in a DMS, undetected bit flips can lead to wrong query results. The detection performance of error codes is typically compared using the codes' minimal Hamming distance $d_{\text{H,min}}$, which is the smallest difference in the bit-wise representation between any two of a code's code words. For error-correcting codes, also the Hamming sphere is important, which for a valid code word includes those invalid code words around it which can be corrected (cf. Equations (3.22) and (3.23)). The downside of correction, however, is that many more multi-bit flips lead to SDC. This is when a multi-bit flip leads from a valid code word into the correction sphere of another valid code word, which then leads to decoding errors. Anyhow, the minimal Hamming distance is obtained through the distance distribution $K^{\mathbb{C}}$ (Equation (3.24)), which maps to any code word weight b (number of bits set to one) the number of code words of that weight. From that we can derive the conditional SDC probability $\varphi_b^{\mathbb{C}}$ that a b-bit flip will not be detected. In this thesis, we are interested not only in the minimal Hamming distance, but we want to compare the codes across their whole distance distributions to get a better understanding how "good" they are in the presence of all possible bit flips. For original Hamming codes there are weight enumerators which directly tell the weight distribution from which we can then derive the distance distribution. From these, we can easily derive the distributions for extended Hamming codes, whereas for shortened Hamming codes this is not as trivial and the weight distribution must be counted. We showed examples for the final distance distributions and conditional SDC probabilities in Figure 3.7. Then, we described a methodology towards computing the distance distribution for XOR checksums, because in the literature we found no concrete descriptions of obtaining it, but only vague expressions like they detect all odd numbered bit flips. In contrast to Hamming codes, the checksum width is typically fixed and the number of data words ($\#\mathbb{D}$) over which the checksum is computed may greatly vary. Using our methodology, we can iteratively compute the distributions for 1-, 2-, and 3-bit checksums. We used weight distribution matrices and triangles to find patterns and rules for computing the distribution for arbitrary $\#\mathbb{D}$. For AN codes, we currently must exhaustively compute the distance distribution in a brute force manner, because due to the convolution of the multiplication (the carries being unpredictable), AN codes do not exhibit such nice structures like Hamming or XOR. Since the distance distribution for parameter A depends on the data width, the brute force approach must be applied to all desired data widths and parameter widths. The problem is that the complexity increases by about a factor of four with each additional data bit and by factor of 2 with each additional bit for A. This becomes very costly especially for larger data widths and we showed that by using a grid approximation method, we could reduce the runtime for a single A and 32-bit data from 3.4 years down to 5 minutes, using a 4-GPU cluster in both cases. This is more than 5 orders of magnitude faster. From all of the measured As, we still need to filter the desired ones and therefore we defined two possible optimality criteria, which select those *golden* As listed in the appendix in Tables A.1 to A.4. We might only be interested in the smallest As which guarantee to detect a minimal bit flip weight and we gathered this set in Table 3.6. Finally, we also showed how to employ the conditional probabilities $\varphi_b^{\mathbb{C}}$ in concrete error models for equal probabilities for all bit flip weights and patterns, as well as the binary symmetric channel (BSC), a well-known error model from information theory. The latter shows that for an error probability of $p = 10^{-6}$, the AN code using our golden A with the same space overhead as Hamming codes exhibits an SDC probability which is almost 13.5 orders of magnitude better than that of the respective Hamming code. Our methods can be used to compute even larger parameters A than we did until now ($1 \leq |A| \leq 16$) and also to pick golden As for future concrete hardware error models.

3.3 THROUGHPUT CONSIDERATIONS

DMSs try to exploit as many hardware features to improve query throughput and latency. Modern CPUs provide instructions which do one computation on multiple data units in parallel, in one (or a few) cycles, which is known as SIMD processing. This SIMD concept is also referred to as *vectorized* processing, because one operation is applied to a *vector* of data units. We will use the terms vectorized processing, *vector instructions*, and *vectorization* to refer to this type of computation. For main memory-centric DMSs, vectorization is a key technology for improving

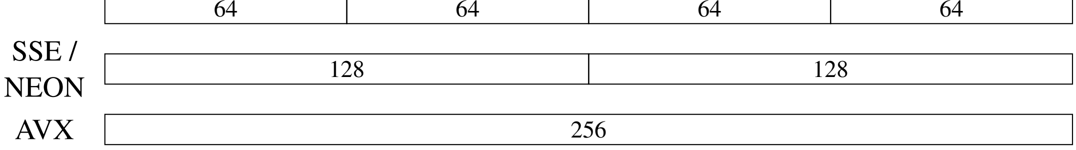


Figure 3.17: Width of Scalar data and SIMD vectors in bits.

query latency and throughput. This has become the standard to improve single-thread performance. Therefore, in this Section we will consider the vectorizability of the three codes as a crucial part of fulfilling the efficiency Requirement $\mathcal{R}2$.

Lots of research was conducted on exploiting these features in DMSs [37, 105, 124, 139, 143, 144, 162, 175, 189, 190, 195] in order to speed up query processing, which is why we examine the selected error codes for vectorizability, i.e. in how far coding operations can be vectorized. There are different instruction sets available, which differ both in the available operations, the vector size, i.e. how many data units are processed in parallel, and the supported data unit widths [74], which is visualized in Figure 3.17. Among the well-known SIMD ISAs are Streaming SIMD Extensions (SSE)[5], Advanced Vector Extensions (AVX) and its extension AVX2, AVX-512, and arm® NEON™ [9]. SSE and NEON™ operate on 128-bit vectors, while AVX and AVX2 use 256-bit vectors, and AVX-512 again doubles the vector width to 512 bits (cf. Figure 3.17). In the following, we will show which operations on Hamming Codes, checksums, and AN codes can be vectorized. We will assume x86 extensions like SSE (up to 4.2) and AVX (1, 2, or 512), but the ideas can in general be mapped to other SIMD instruction sets like arm®NEON™.

3.3.1 Test Systems Descriptions

In the following, we will conduct several microbenchmarks so we take the opportunity here to present the two test systems which we use throughout this thesis. We measure all following benchmarks for the two systems from Table 3.7, where System 1 is one of the first Intel® Skylake CPUs and the latter being an Intel® Xeon Phi™ standalone CPU. Both systems have quite different hardware characteristics, where important differences are the number of cores (4 vs. 68), hyper-threads (hardware threads per core, 2 vs. 4), the different base frequencies (2.7 GHz vs. 1.4 GHz) and the different memory architectures and core interconnects. Also, the actual core architectures are quite different. While both CPUs exhibit out-of-order execution, the Xeon Phi is based on much weaker Silvermont Atom cores[6]. For both systems, we explicitly disable the Intel™turbo boost feature and force the operating system's scaling governor to `performance` for more stable results and better comparability. We will conduct all benchmarks on both systems to provide runtime numbers for two very diverse compute architectures. As we will see on several occasions, we measured quite different runtime behavior for these two hardware platforms.

3.3.2 Vectorizing Hamming Coding

For hardening and error detection, software Hamming codes require only a small set of instructions, namely bit-wise AND for extracting bit patterns from the original data (masking), bit population count (*popcount*) to compute the actual parity bits, and comparison of the stored and newly computed code bits. Since Hamming is a systematic code, for softening the original data can be simply loaded or copied. To require as little operations as possible, the original data is stored successively as individual vectors and the Hamming code bits (or bytes) are also stored in a single block for each data vector. By that, data processing operations like addition, aggregations, or filtering can be applied easily on the data bits. For data modification, the code bits can be easily updated as well since they are stored separately.

[5] There are several versions: SSE, SSE2, SSE3, SSSE3, SSE4, SSE4a and SSE5, which we simply refer to as SSE.
[6] `http://vrzone.com/articles/xeon-phi-knights-series-continues-landing-2015/64112.html`

Category	Type	System 1	System 2
CPU	Name	Core i7-6820HK	Xeon Phi 7250
	Codename	Skylake	Knights Landing
	Frequency	2.70 GHz	1.40 GHz
	Cores	4	68
	Threads per Core	2	4
	Total # Threads	8	272
	L1i-Cache	32 KiB$^{(pc)}$	32 KiB$^{(pc)}$
	L1d-Cache	32 KiB$^{(pc)}$	32 KiB$^{(pc)}$
	L2-Cache	256 KiB$^{(pc)}$	1024 KiB$^{(2c)}$
	L3-Cache	8192 KiB$^{(sh)}$	—
	Core Interconnect	Ring Bus	Mesh
	Vector ISAs	SSE*,AVX-2†	SSE*,AVX-2†,AVX-512‡
Main Memory	Type	— DDR4 —	
	Size	2×8 GiB $= 16$ GiB	6×32 GiB $= 192$ GiB
	Frequency	2133 Mhz	2400 MHz
Software	OS	Ubuntu 16.04	
	Compiler	GCC 7.3	GCC 7.0
	Turbo Boost	off	
	Scaling Governor	performance	

Table 3.7: Basic Specification of the 2 measurement systems. DRAM frequencies are reported by `lshw`. CPU Vector ISAs are reported by `lscpu`. $^{(pc)}$: per core. $^{(sh)}$: shared across all cores. $^{(2c)}$: shared by 2 cores (1 tile). *: SSE, SSE2, SSSE3, SSE4.1, and SSE4.2. †: AVX and AVX-2. ‡: AVX-512F, AVX-512PF, AVX-512ER, AVX-512CD.

The crucial point stays the generation of the Hamming code bits. For hardening and detection, the AND operation is supported for whole vectors, while bit counting is currently only supported for single values and must be done with right-shifts, logical ANDs, additions and subtractions [182, Section 5.1, p. 82], all of which are in turn supported by modern server-grade CPUs' SIMD instruction sets. For instance, Mula *et al.* show population count vectorization using AVX2 [124] and they also discuss population count algorithms in much more detail than it is necessary here. Some of the presented approaches use table look-up to accelerate population counting. When such tables are rather big, the problem is that such a table resides in main memory which can become corrupt over time and we would need methods to maintain these tables' integrity, i.e. bit flip detection and correction. Another difference to their setting is, that we need not compute population counts of long bitmaps, but compute individual popcounts for the parity bits, i.e. multiple popcounts per Hamming code word. Consequently, there are many more instructions besides the population counting and considerations for amortization over long bitmaps do not apply. We will now first show how we vectorize popcount, compare 3 different approaches, and then we show how the Hamming coding itself is vectorized.

Vectorized Population Counting

As Mula *et al.* note, there are basically 2 efficient population count techniques besides the hardware based `popcount` instructions [124]. For the latter there is, however, currently no vectorized variant available. Figure 3.18 visualizes the two techniques' concepts. The first one is a binary *tree adder*, shown in Figures 3.18a and 3.18b. It works by a recursive pattern, which adds together neighboring counts. It starts with considering each single bit as a count on its own and can be described as follows:

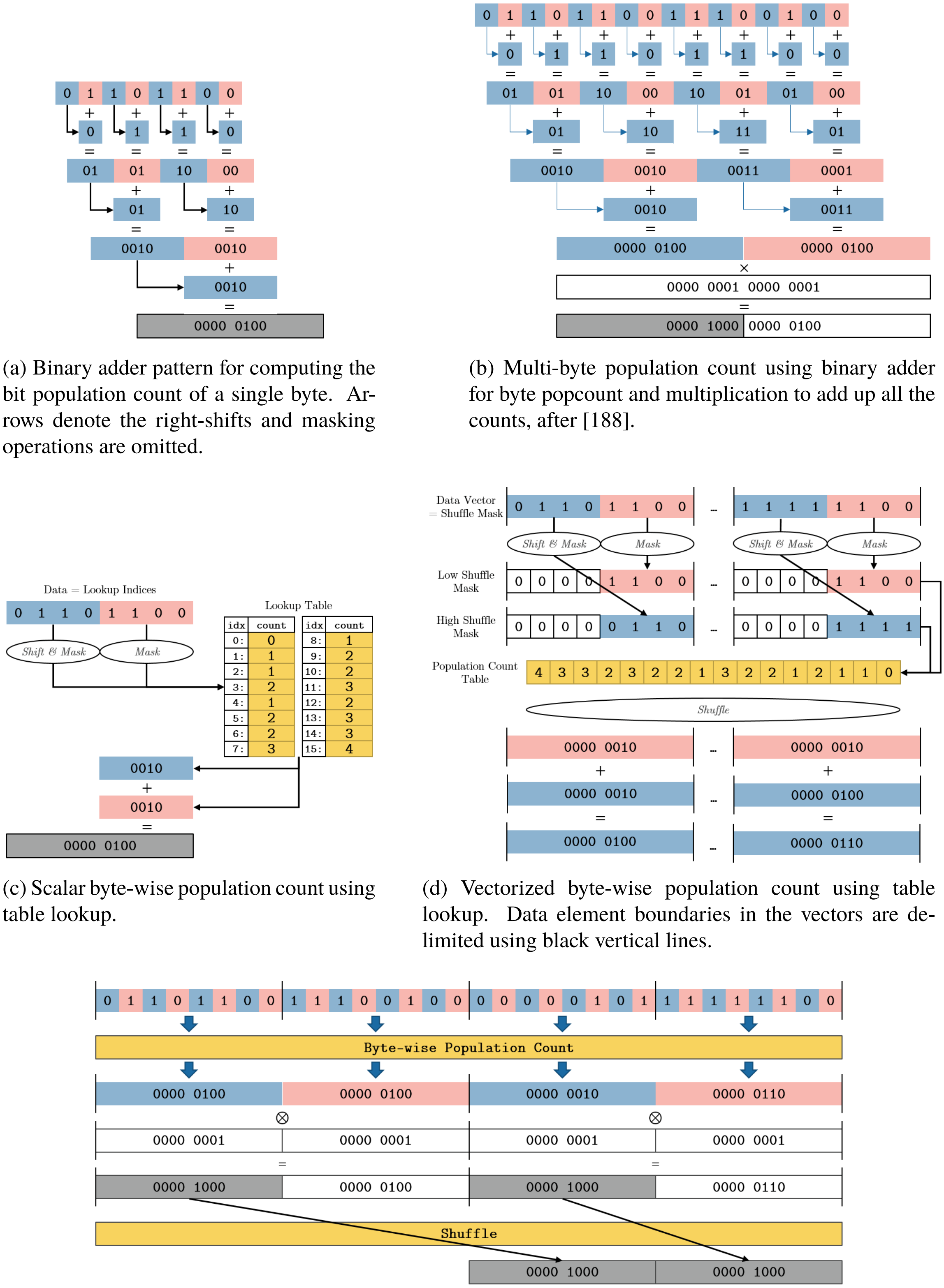
1. Variable `data` contains the element for population counting.
2. Set `shiftNum` to 1.
3. While `shiftNum` $<$ data bit width, do:
    (a) Shift `data` right by `shiftNum` number of bits and store the result in variable `shifted`.
    (b) Retain only the bits of interest for the original data and `shifted` by masking out all other bits.
    (c) Add together masked `data` and masked `shifted` and store the result again in `data`
    (d) Double `shiftNum`
4. The rightmost $log_2|\texttt{data}|$ bits contain the bit population count.

Figure 3.18a visualizes the concept behind the binary tree adder for an 8-bit value. The right-shifts are denoted by the arrows while $+$ symbolizes the addition of the sub-sums, which are color coded blue and red. The 8-bit case is straight forward and is the basis for the larger data widths: Figure 3.18b consists of two parallel 8-bit tree adders and the counts for the bytes are added together with multiplying by the bit pattern `0000 0001 0000 0001`, so that the total population count is found in the *most significant* byte. Here, $\times$ denotes the multiplication to add up individual byte-population counts. Implementations for 32- and 64-bit adders are the same except that the pattern is expanded appropriately to multiple "`0000 0001`"-bytes.

The second approach, proposed by Muła, looks up the population counts for 4 bits using a very small lookup table [123], as depicted in Figure 3.18c. As we noted above, storing lookup tables in main memory makes them vulnerable to bit flips, but such a small table can easily be recomputed periodically or just before iterating over arrays of Hamming code words. It follows a `0 1 1 2`-pattern which can be applied recursively on itself to generate larger patterns, which results in the lookup tables from Figures 3.18c and 3.18d. In Figure 3.18c, the 4 upper and lower bits are separately used as index to fetch the precomputed popcount from the lookup table. The resulting counts are added up afterwards. In Figure 3.18d, the lookup table (denoted as Population Count Table) is stored in a vector and the actual data vector is used as a shuffle mask to shuffle the population count table units around. As in Figure 3.18c, this is done separately for the lower and upper 4 bits of each byte in the vector element and the counts are added after the shuffling.

Figure 3.18e generalizes both techniques to vectorized multi-byte population count[7]. It starts with vectorized byte-wise popcount which can be implemented using either of the two discussed techniques and it then uses the multiplication improvement from Figure 3.18b. All steps are vectorized and here the final shuffling step is explicitly included so that all population counts are packed together. More details are given in Appendix A.2.2. In the following we will determine which is the faster of the two techniques, to decide which one to use for Hamming coding.

[7] On more modern hardware platforms, vectorized population count may be available like `VPOPCNT` in Intel® Knights Mill processors, which is however supposed to only support population counting on 32-bit and 64-bit integers, so that for smaller integers a fallback may be faster than additional shuffling operations to align them to 32-bit boundaries.



(a) Binary adder pattern for computing the bit population count of a single byte. Arrows denote the right-shifts and masking operations are omitted.

(b) Multi-byte population count using binary adder for byte popcount and multiplication to add up all the counts, after [188].

(c) Scalar byte-wise population count using table lookup.

(d) Vectorized byte-wise population count using table lookup. Data element boundaries in the vectors are delimited using black vertical lines.

(e) Generic vectorized population count with all steps being vectorized. Byte-wise population count is implemented using (a) or (d) and the multiplication is done on input size granularity.The final shuffling packs together the popcounts which reside in the elements' highest bytes due to the multiplication. Data element boundaries in the vectors are delimited using black vertical lines.

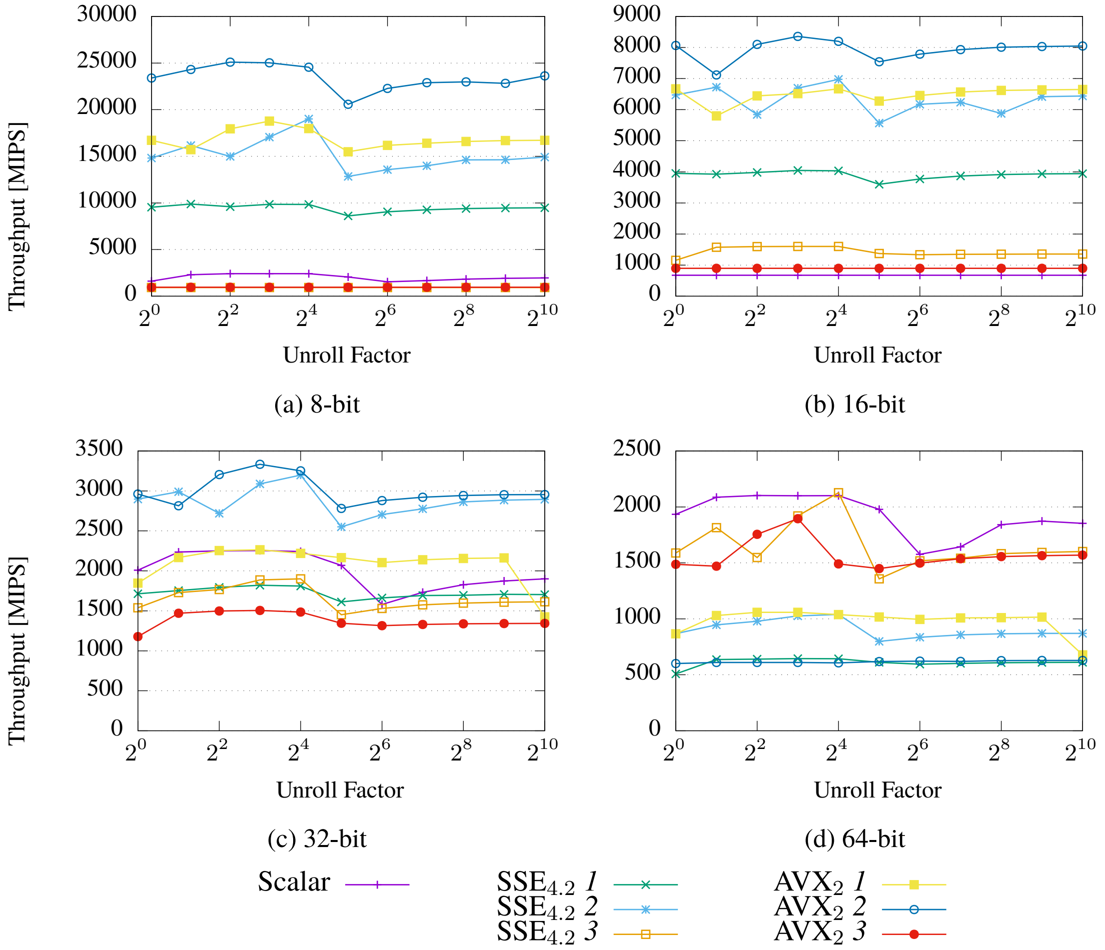
Figure 3.18: Population counting using binary adder for single byte integers (a) and multi-byte integers (b), and byte-wise popcount using table lookup as scalar (c) and vectorized (d) variants. (e) generalizes the concepts to vectorized multi-byte population counting. Neighboring sub sums are colored alternating with blue and red background and the final result is colored gray.

We measured the throughputs of the individual scalar and vectorized population counting variants. For system 1, the results are shown in Figure 3.19, showing throughput in million integers per second (MIPS). For the cases 8-bit (a), 16-bit (b), and 32-bit (c), we see that the *second* variants of SSE4.2 and AVX2 provides the highest throughputs for each ISA family. The throughput is dramatically reduced with increasing bit widths. For AVX2 and SSE4.2, respectively, in the 8-bit case from around 25 000 (unroll factors 4 and 8) and around 18 800 (factor 16) MIPS (Figure 3.19a), down to about 8300 (factor 8) and 6700 (factor 16) for the 16-bit case (Figure 3.19b) and down to 3300 (factor 8) and 3200 (factor 16) MIPS for the 32-bit case. For 64-bit integers (Figure 3.19d), SSE4.2 variant 3 is fastest with factor 16 and slightly more than 2100 MIPS, otherwise the scalar code is the fastest with also around 2100 MIPS (factors $2 \dots 16$). Table 3.8 lists the highest throughput numbers and the respective best unroll factors for system 1. There we can see that typically, the best unroll factors are typically between 2 and 16, with only a few exceptions. Additionally, the cells representing the fastest variant per bit case are highlighted in gray. The population counting is only a sub-operation for Hamming hardening and we will verify shortly whether for the coding itself these throughputs translate into respective ratios for hardening, as well.

We also measured the throughputs of the individual scalar and vectorized population counting variants for system 2, shown in Figure 3.20. We see that throughputs differ greatly from those for system 1. The results are shown in Figure 3.20. For some bit cases, the best throughput for system 2 is even an order of magnitude smaller than for system 1. For the cases 8-bit (a), 16-bit (b), and 32-bit (c), we see that the *first* variants of SSE4.2, AVX2 and AVX512 provide the highest throughputs for each ISA family. For 64-bit (d), again the third variant of each vectorization family is fastest, as for system 1. However, here, the scalar variant is already faster than the vectorized ones since the 32-bit case. The throughputs are, again, dramatically reduced with increasing bit widths, as can be seen in Table 3.9.

Vectorized Hardening

In the following, we will discover in how far the population count variants have an influence on Hamming coding. The Hamming coding itself is simply the calculation of the even parity for several data bit subsets. The schema for vectorized Hamming coding is depicted in Figure 3.21: On the input data vector all patterns for selecting the data bit subsets are applied one after another, where patterns are appropriately as in Figures 3.3 and 3.4. On each of the obtained data subset vectors vevctorized popcount is applied as noted above. Afterwards, the even parity is obtained for each data element's population count (for each subset vector) by selecting the LSB. The individual parity bits are interleaved so that for each data element the code bits are consecutively stored in their own bytes. Assuming a vector operation can operate on k elements in parallel (k depends on the data width), then k Hamming code words can be computed in parallel in a single hardware thread. Each *byte* in the code bits part (bottom of Figure 3.21) contains the code bits of a single data element of the original data vector. So finally, first the whole data vector is stored and then all code bits arranged in individual bytes. As can be seen in Figure 3.4, a 64-bit data word requires an additional 8 code bits for an Extended shortened Hamming code, so that a single byte suffices. A concrete implementation is provided in Appendix A.2.2. We now show that the speed of the population counting correlates with the Hamming coding throughput *and* that, as for the population count, the fastest variant depends on the actual hardware platform. Additionally, this proves that the selected method is indeed a very fast one and it allows to ultimately give a fair comparison later against the other coding schemes.



(a) 8-bit

(b) 16-bit

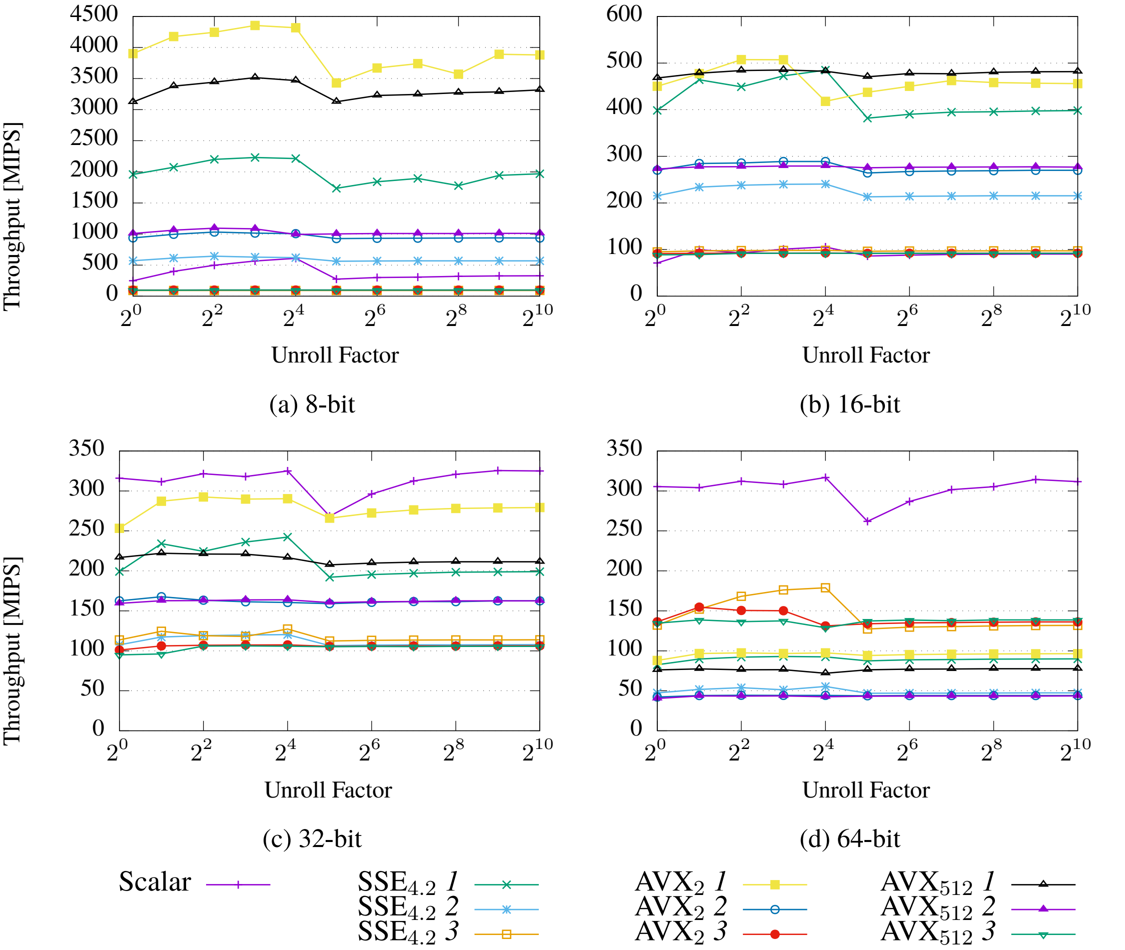
(c) 32-bit

(d) 64-bit

Figure 3.19: Comparison of the different population count variants for system 1.

Bit		Scalar	$SSE_{4.2}$			AVX_2		
Case			*1*	*2*	*3*	*1*	*2*	*3*
8	Highest MIPS	2416.6	9876.9	19 004.9	936.2	18 778.8	25 095.8	952.2
	@ Unroll Factor	16	2	16	32	8	4	1024
16	Highest MIPS	673.0	4045.9	6976.3	1601.8	6670.5	8354.7	896.9
	@ Unroll Factor	64	8	16	16	16	8	32
32	Highest MIPS	2253.6	1819.9	3198.5	1900.3	2263.9	3333.3	1504.7
	@ Unroll Factor	8	8	16	16	8	8	8
64	Highest MIPS	2102.2	644.3	1039.5	2126.8	1058.7	628.3	1894.8
	@ Unroll Factor	4	8	16	16	8	1024	8

Table 3.8: Best population count throughputs and unroll factors for system 1. Best variants per SIMD variant and bit case are highlighted in gray.



(a) 8-bit

(b) 16-bit

(c) 32-bit

(d) 64-bit

Figure 3.20: Comparison of the different population count variants for system 2. Best variants per SIMD variant and bit case are highlighted in gray

Bit Case		Scalar	$\text{SSE}_{4.2}$ *1*	*2*	*3*	AVX_2 *1*	*2*	*3*	AVX_{512} *1*	*2*	*3*
8	Highest MIPS	608.5	2230.0	643.6	92.1	4353.9	1031.9	99.1	3514.8	1092.9	98.6
	@ Unroll Factor	16	8	4	8	8	8	16	8	4	8
16	Highest MIPS	105.6	484.6	240.4	98.3	507.3	289.0	92.7	485.0	279.2	92.1
	@ Unroll Factor	16	16	16	16	4	16	16	8	16	16
32	Highest MIPS	325.4	242.2	120.3	127.3	292.5	167.7	107.5	222.1	163.9	106.1
	@ Unroll Factor	512	16	16	16	4	2	16	2	16	8
64	Highest MIPS	316.8	92.9	55.4	178.8	97.5	44.4	154.7	77.6	43.6	138.7
	@ Unroll Factor	16	8	16	16	4	4	2	512	2	2

Table 3.9: Best population count throughputs and unroll factors for system 2.

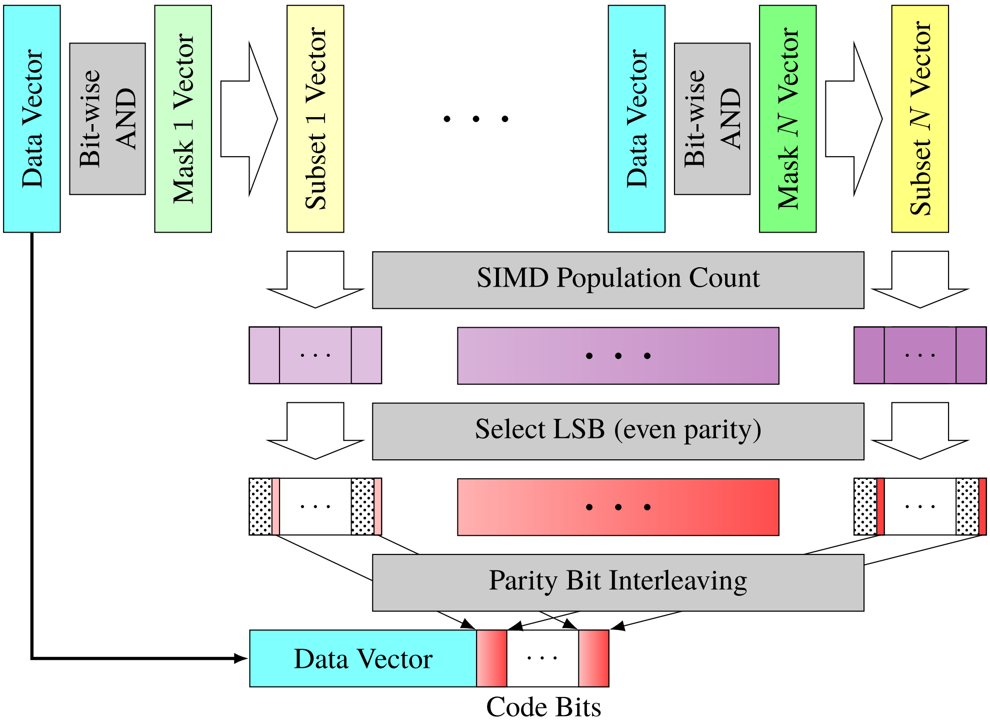


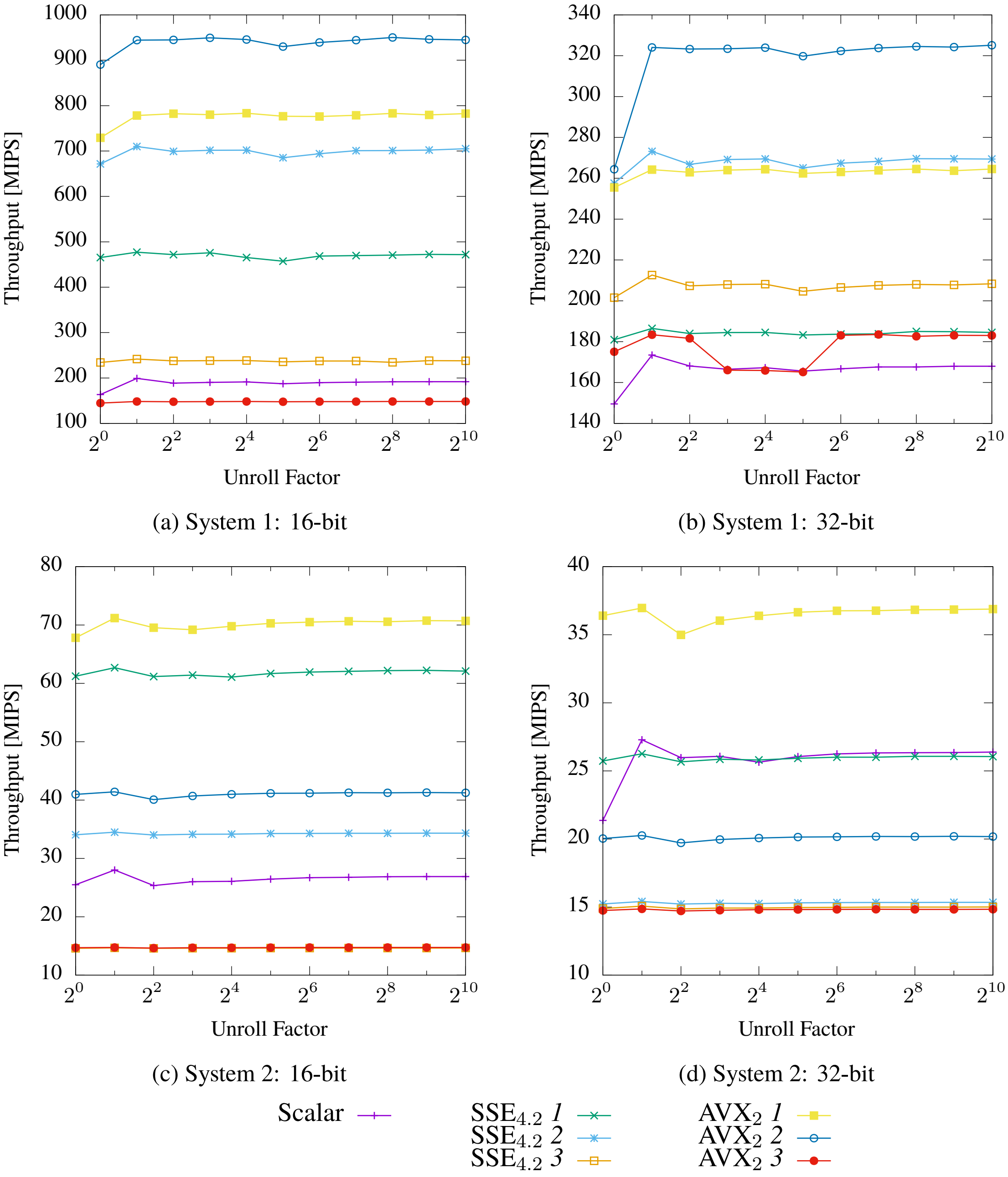
Figure 3.21: Schema of vectorized Hamming coding with all steps being data-parallel.

We test all of the three discussed variants:

1. Binary adder as in Figures 3.18a and 3.18b.
2. Lookup table variant as in Figures 3.18c and 3.18d.
3. Scalar `popcnt` instruction with vector load/store and element-wise `extract`.

As a baseline we include the purely scalar variant which only uses the built-in hardware instruction `popcnt` and we prohibit automatic compiler vectorization through the compiler flag `-fno-tree-vectorize` while we otherwise use the same settings as in Section 3.4.2. We use automatic compiler-based loop unrolling for powers of 2, to see whether this is beneficial for Hamming coding. Again, we use both measurement systems summarized in Table 3.7. Figures 3.22a and 3.22b show the throughput on system 1 in MIPS for 16-bit (a) and 32-bit data (b). There, we first see that the automatic, compiler-based loop unrolling brings only little to no gains beyond a value of two. Second, we can see that the vectorized binary adders (1) and lookup variants (2) greatly improve on the scalar variants. Variant 3 only uses vectorized load / store instructions and the AVX variant is even *slower* than the scalar one for 16-bit data. On system 1, for 32-bit data variant 3 is mostly faster than scalar for both SSE and AVX. Variant 1 (binary adder) improves on scalar by 110 % (SSE) and 194 % (AVX), while variant 2 (lookup) improves on scalar by 154 % (SSE) and 243 % (AVX). Comparing variants 1 and 2 with each other on system 1, this means the lookup variant (2) improves Hamming coding over the binary adder (1) by 16 % (SSE) and 17 % (AVX) for 16-bit data and by up to 21% (SSE) and 16 % (AVX) for 32-bit data.

Figures 3.22c and 3.22d show the throughput for system 2, again in MIPS, for 16-bit (c) and 32-bit data (d). Again, the compiler-based loop unrolling has mostly only little to no effect. Furthermore, the throughput numbers are about an order of magnitude lower than for System 1, which can be



(a) System 1: 16-bit

(b) System 1: 32-bit

(c) System 2: 16-bit

(d) System 2: 32-bit

Figure 3.22: Comparison of Hamming coding with different population count implementations in MIPS on system 1 (upper two graphs) and systems 2 (lower two graphs).

largely attributed to the much lower core frequency and older core architecture. However, the most important observation is that the binary adder now wins over the lookup approach and variant 3 for both SSE and AVX is substantially slower than the scalar variant. The binary adder (variant 1) is 102 % (16-bit, SSE), 212 % (16-bit, AVX), and 18 % (32-bit, AVX) faster than scalar execution. The picture is even more obscure for 32-bit data: The throughput for variants 1 and 2 is almost cut in half (compared to 16-bit), while the scalar variant and variants 3 almost stay the same. Still, the binary adder is superior but the scalar code is now the second fastest. Consequently, on this architecture the binary adder (variant 1) should be used. Finally, the best unroll factors are all exactly 2.

The measurements showed that the best variant depends on the actual hardware platform and we will respect this in the upcoming comparison benchmarks.

3.3.3 Vectorizing XOR Checksums

As for Hamming coding, XOR checksums only require the creation (and comparison) of the checksum over a block of data for hardening and error detection and since XOR checksums are also systematic, softening is also simply reading or copying the original data. Checksum creation can be easily vectorized as the XOR operation is available on whole vector widths for all SIMD instruction sets (SSE, AVX, neon). A block of data is therefore interpreted as an array of vectors which are successively XORed onto a zero-initialized checksum vector. This could in principal be done in a tree-binary adder fashion or just sequentially, both of which have the same number of XOR operations. When it is required to create checksums that are shorter than the vector width, the tree adder method can be used to further condense it. This is a bit different from the method presented for Hamming coding in that it does no shifting and masking of the individual bits, but only full additions. For instance, from a 128-bit vector, a 32-bit checksum can be obtained by XOR-ing first its higher an lower 64 bits and then that intermediate's higher and lower 32 bits. Table 3.10 lists the instructions used for computing XOR on different SIMD instruction sets.

Since checksums operate on larger chunks of data, processing or modification operations and the checksum generation, for potentially both old and new data, could be done simultaneously. After loading a vector, it is XOR-ed against the checksum accumulator vector, then the data is processed or modified, whereas for the latter an additional accumulator for the new checksum is used. At the end of a data block, the generated checksum is compared against the stored one and for modification the checksum of the modified data is stored. However, the problem that bit flips are detected only after the whole checksum is computed, remains and results from the processing or modification operation must be undone or discarded.

SIMD ISA	vector width [bits]	instruction[8]	GCC intrinsic[9]
SSE	128	`pxor`	`_mm_xor_si128`
AVX2	256	`vpxor`	`_mm256_xor_si256`
AVX-512F	512	`vpxord`	`_mm512_xor_si512`
NEON	128	`veor`	e.g. `veorq_u16`

Table 3.10: XOR specific instructions and compiler intrinsics for different SIMD ISAs.

SIMD ISA	vector width [bits]	data width [bits]	instruction[8]	GCC intrinsic[9]
SSE	128	16	`pmullw`	`_mm_mullo_epi16`
		32	`pmulld`	`_mm_mullo_epi32`
AVX	256	16	`vpmullw`	`_mm256_mullo_epi16`
		32	`vpmulld`	`_mm256_mullo_epi32`
AVX-512BW		16	`vpmullw`	`_mm512_mullo_epi16`
AVX-512F	512	32	`vpmulld`	`_mm512_mullo_epi32`
AVX-512DQ		64	`vpmullq`	`_mm512_mullo_epi64`
NEON	128	{8|16|32}	`vmul.i{8|16|32}`	`vmul_{u|s}{8|16|32}`

Table 3.11: Multiplication specific instructions and compiler intrinsics for different SIMD ISAs.

3.3.4 Vectorizing AN Coding

In contrast to Hamming and XOR, AN coding requires different operations for hardening, softening, and error detection, namely multiplication (Equation (3.10)), division (Equation (3.11)), and modulus (Equation (3.12)), respectively. From these three, only multiplication is directly available as vectorized instruction, where some available instructions and compiler intrinsics are shown in Table 3.11. In contrast, division is only available for single- and double-precision floating point numbers (for SSE and the like, but not for arm® NEON™), and the modulus operation has no vectorized counterpart at all [74]. For division, we convert the integers to the next larger floating point type (floats), however, for SSE, AVX and the like, there are only conversion instructions for 32- and 64-bit data widths to single- and double-precision floats. Care must be taken to not loose precision, which can happen when e.g. converting from large 64-bit numbers to both single- or double-precision floats.

3.3.5 Summary and Conclusions

Vectorization is a crucial technique to improve query throughput and latency in main memory-centric DMSs. As we have shown in this Section, the coding operations of all three code families can be (to some degree) vectorized. For Hamming codes, we only need to vectorize the computation of the code (parity) bits. There, each individual code bit can be computed in parallel for several data words. For that, the population count is needed, since each code bit is a parity bit over a subset of the data bits. Since there is currently no vectorized population count instruction available in the considered vector ISAs, we investigated three different alternative implementations and measured throughput numbers on our two evaluation systems. As we could see, the fastest method depends on the underlying hardware. However, from a reliability perspective, not all of these techniques are desirable. For instance, when using lookup tables, these need to be stored in unreliable main memory and therefore they must somehow be hardened against bit flips, which goes beyond the scope of this thesis. For XOR checksums, the situation is much better, because the considered vector ISAs all support XOR instructions over whole vector registers. Finally, for AN coding only the hardening (multiplication) can be vectorized, while for error detection (modulo) and softening (division), there are no vector integer instructions available.

[8] For SSE/AVX see `https://software.intel.com/sites/landingpage/IntrinsicsGuide`. For NEON™ see `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0489e/CJAJIIGG.html`

[9] For SSE/AVX see `https://software.intel.com/sites/landingpage/IntrinsicsGuide`. For NEON™ see `https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/ARM-NEON-Intrinsics.html`

3.4 COMPARISON OF ERROR CODES

We previously presented the three coding schemes Hamming codes, XOR checksums, and AN codes separately. Now, we compare them in accordance with our Requirements $\mathcal{R}1$ to $\mathcal{R}3$, i.e. their *effectiveness* in Section 3.4.1, their *efficiency* in Section 3.4.2, and their *runtime adaptability* in Section 3.4.3.

3.4.1 Effectiveness

Requirement $\mathcal{R}1$ dictates that our error detection solution should be *effective*. The effectiveness of a code comprises several aspects: 1. detection granularity (e.g. data block or data unit granularity), 2. SDC probabilities (summary and comparison of findings from Section 3.2), and 3. instruction coverage (the ability to detect errors induced when a CPU executes an instruction), whereas the first two influence each other. We will compare first the detection granularity, then the detection capabilities, and finally instruction coverage.

Detection Granularity

All three techniques can allow bit flip detection on value granularity, but (XOR) checksums are usually employed for larger blocks of data, e.g. whole disk blocks [59, 170]. For single-value detection, when the XOR word size is as large as the data word, this ends up in data duplication. *Hardware* Hamming (e.g. ECC main memory) has a constant data block width (64 bits) and is thus oblivious to the actual data size, while *software* Hamming can be adjusted appropriately to the actual data width, or it can also be tailored to various blocks of data. AN coding is by nature designed to work on (integer) value granularity. Although it could be applied to blocks of arbitrary length, e.g. by the use of an arbitrary precision math library [52, 56, 153], this is not feasible due to the expensive determination of its parameter A (cf. Section 3.2.3). For Hamming and AN, when covering multiple data units, the SDC probabilities are then "shared", which then can greatly increase the SDC probability of the individual data units.

Detection Capabilities

Hamming codes detect all 1- and 2-bit flips, but cannot detect all 3-bit flips, because although they have $d_{\mathrm{H,min}} = 4$, the correction does not allow to detect any 3-bit flip which leads into the correction sphere of another code word. Figure 3.7 shows that all Hamming codes exhibit a zig-zag pattern for their SDC probabilities. Those for *even* bit flip weights (BFWs) are comparably low with probabilities in the range of $0.13\% \ldots 7.7\%$ for all shown data widths. Those for *odd* BFWs are much worse with probabilities between $48.2\% \ldots 100\%$, i.e. for some data widths some BFWs are always undetectable. For instance, the $(13, 8)$ Hamming code can not detect any code word inversions. For the $k = 24$ Hamming code, the probabilities are worse than for the other codes and this shows that when we increase k such that no additional code bit is used for Hamming, the SDC probabilities become worse.

XOR checksums can detect *any* odd BFWs, independent of the checksum width. Regarding even BFWs, we see various effects: 1. All checksums widths exhibit comparably high SDC probabilities for 2-bit flips and the chances decrease afterwards. 2. For increasing checksum width, the chances of SDC are reduced in total. 3. In contrast, for increasing $\#\mathbb{D}$, the chances of SDC are mostly *increased*. 4. For odd $\#\mathbb{D}$ inversion cannot be detected. 5. Finally, for even $\#\mathbb{D}$, the last $|\mathbb{X}|$ BFWs are always detected. In summary, XOR checksums principally perform better with larger checksum widths, the fewer data units they cover, and when they cover an even number of data units.

The SDC probability of an AN code depends on the data width and the parameter width. Larger data widths require larger As to achieve the same minimal detected bit flip weight η (cf. Table 3.6). Increasing the bit width of A also in principal reduces the SDC probabilities. The probability distributions form a jigsaw pattern for As of the same bit width, where increasing As have a flatter curve in the middle.

All three codes exhibit specific patterns regarding their SDC probabilities. Hamming and XOR exhibit zig-zag patterns, where Hamming can only detect up to double bit flips reliably, while XOR can detect reliably only all odd numbered BFWs. Furthermore, XOR and AN codes show anomalies at the low and high BFW ends. While these are fixed and predictable for XOR checksums, for AN codes we can find *golden* As which have optimal error detection behavior. The SDC probabilities for Hamming are the worst, because it is an error correcting code, where many bit flips lead into the correction sphere around code words. Generally, the larger the correction sphere is, the worse such a code performs in comparison to detection-only.

At the end of Section 3.2 we gave examples for concrete (unconditional) SDC probabilities of $(13, 8)$ Hamming and AN codes under the BSC error model. Now, In Figure 3.23, we compare the *conditional* probabilities of the data widths $k \in \{8, 16, 24, 32\}$ bits. The unconditional probabilities could be obtained the same way as in Section 3.2, but using the conditional probabilities better shows the probabilities of the AN codes. p_b^H is the probability for the Hamming code and the $p_{b,\eta}^A$ are the ones for different AN codes, with the appropriate As taken from Table 3.6. For every subgraph, the AN code with the smallest η $(= 2)$ has the *same* code width as the respective Hamming code. All subgraphs show that with additional code bits, the AN codes reduce the probabilities of SDC and simultaneously increase the minimal detectable bit flip weight (η).

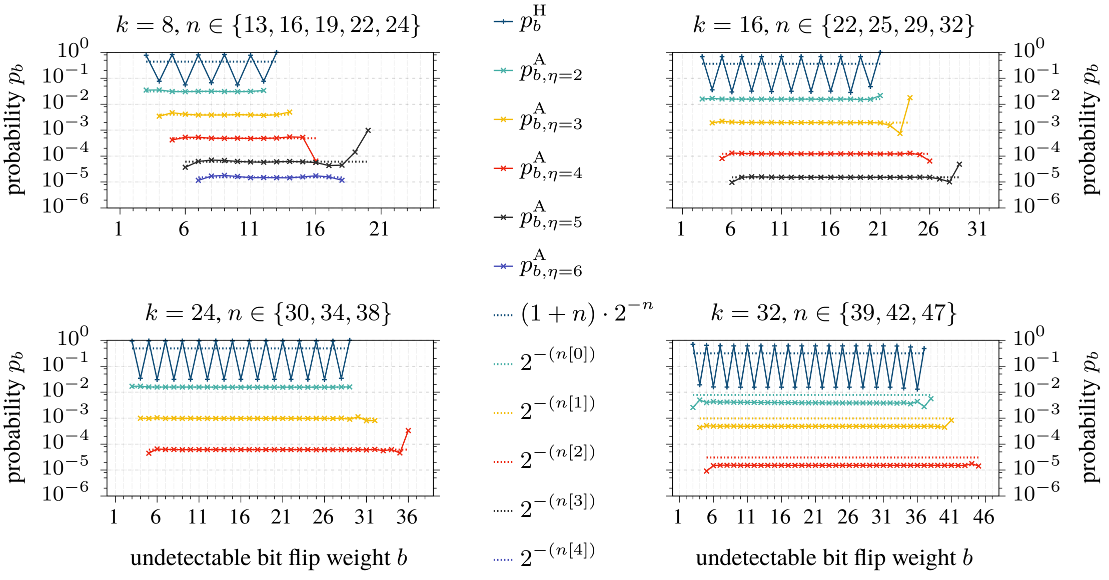


Figure 3.23: Comparison of unconditional SDC probabilities for Hamming an AN.

Instruction Coverage

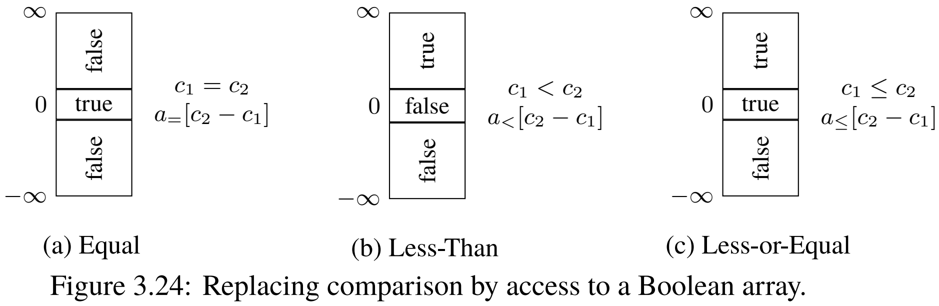
Besides bit flips in data, there may also be errors in a CPU's ALU, so that e.g. arithmetic and logic operations yield wrong results. We first consider arithmetic operations, where the cases of updating an existing value and creating a new one lead to the same observations. Hamming and checksum codes can not detect any instruction errors, because their redundant code information can only be generated *after* an arithmetic operation has finished. By that, if there was a bit flip in an instruction, the redundant code data would be updated on already corrupt data. In contrast, AN codes are arithmetic codes and when applying arithmetic operations onto AN code words, both the new data and the new redundant code information are computed during the very same operation. By that, after any arithmetic operation we could immediately detect bit flips in the result that occurred during the ALU operation. However, this property is not given when overflows occur.

Detecting bit flips in logic operations, like AND-ing or OR-ing, is not possible for any of the three coding schemes. The only exception is the XOR *operation*, which leads to valid code words again for Hamming and XOR checksums, as these are linear codes. Otherwise, the same as for arithmetic operations applies to Hamming and XOR.Furthermore, all of these operations do not lead to valid AN code words.

Error detection for comparison operations like $<, \leq, =, \ldots$ (cf. Equation (3.15)) is much more challenging, since comparisons are required for error detection by all codes. We could use a technique similar to branch-avoidance by replacing the comparison with an array access, as depicted in Figure 3.24. Suppose that for each comparison operator there is an infinitely large array of Boolean values:

$$a_{\mathsf{Op}}[i] \in \{\top, \bot\},\ \mathsf{Op} \in \{<, \leq, \ldots\},\ i \in \{-\infty \ldots 0 \ldots \infty\}.$$

The difference of the two operands yields the position in the array, where we find the Boolean value representing the result of the original comparison operation. For instance, Figure 3.24a illustrates the array contents for the equality comparison. There, we compute the difference of the two operands $c_2 - c_1$ and the array contains only one `true` value at position zero, since $c_2 = c_1 \Leftrightarrow c_2 - c_1 = 0$. Likewise, for '$<$' comparison (Figure 3.24b), we find `true` only for $i > 0$ and for '$\leq$' `true` is at all $i \geq 0$. However, infinite arrays can not be stored in memory and their contents would have to be verified at runtime as well. Consequently, errors in logic operations $(\&, ||, \oplus, \neg)$ cannot be detected by *any* of the codes. Bit operations like AND-ing, OR-ing, or inverting all bits produce *invalid* code words. While for Hamming and XOR checksums the XOR operation on two code words yield another valid code word, this is not true for AN coding. In total, these operations must be protected in another way. Therefore, we assume *reliable* comparison and logic operations. Both areas must be considered separately in future work.



(a) Equal (b) Less-Than (c) Less-or-Equal

Figure 3.24: Replacing comparison by access to a Boolean array.

3.4.2 Efficiency

Requirement $\mathcal{R}2$ dictates that our error detection solution should be *efficient*. Regarding efficiency, we first discuss the memory consumption, and then conduct microbenchmarks for several classes of operations.

Memory Consumption

Hamming codes have a static structure and the number of redundant bits only depends on the data bit width. By that, their memory overhead only depends on the data bit width. They are defined as having $2^m - 1$ code bits, containing m redundancy bits, where $m \geq 2$. Extended Hamming codes add the overall parity bit, which results in 2^m code bits and $m + 1$ check bits. For shortened Extended Hamming codes, the code length lies between two powers of 2, i.e. $2^{m-1} < |\mathbb{C}_{H_{\text{Ext}}}| \leq 2^m$ for $m + 1$ redundant bits and we can derive m by

$$m = \lceil \log_2 |\mathbb{C}_{H_{\text{Ext}}}| \rceil.$$

Consequently, the overhead is

$$\frac{m}{|\mathbb{D}|}.$$

For XOR checksums, the memory consumption overhead depends on the checksum width and the block width. When we can assume that the block size is always a multiple of the checksum width, then the overhead is

$$\frac{1}{\#\mathbb{D}},$$

which is independent of the actual data width.

For AN codes, the memory overhead depends on both the data bit width and the bit width of parameter A. Following, we assume the golden As from Tables A.1 to A.4. The memory overhead is

$$\frac{|A|}{|\mathbb{D}|}.$$

Figure 3.25 shows the memory overheads for all codes in a single graph. There are multiple curves for AN coding with the golden As for different minimal bit flip weights (η). The x axis represents $|\mathbb{D}|$ for Hamming and AN, while it represents $\#\mathbb{D}$ for XOR. We can see that AN with $\eta = 2$ has almost always the same overhead up to data bit width of 32 bits like Hamming, which also has $\eta = 2$. The little bumps in the Hamming curve indicate where an additional redundant bit is added and AN codes seem to have such a bump exactly 2 data bit widths earlier. The curves indicate logarithmic decrease of the memory overhead for all codes, where XOR has by far the least memory overhead which is up to an order of magnitude less than Hamming and AN with $\eta = 2$.

Microbenchmarks Description

For measuring the efficiency of the coding operations, in the following we conduct microbenchmarks[10] where we compare the three codes against each other. We test the following operations:

[10] The source code of the benchmarks is publicly available under `https://github.com/brics-db/coding_benchmark/`. See also `https://brics-db.github.io/coding_benchmark/`

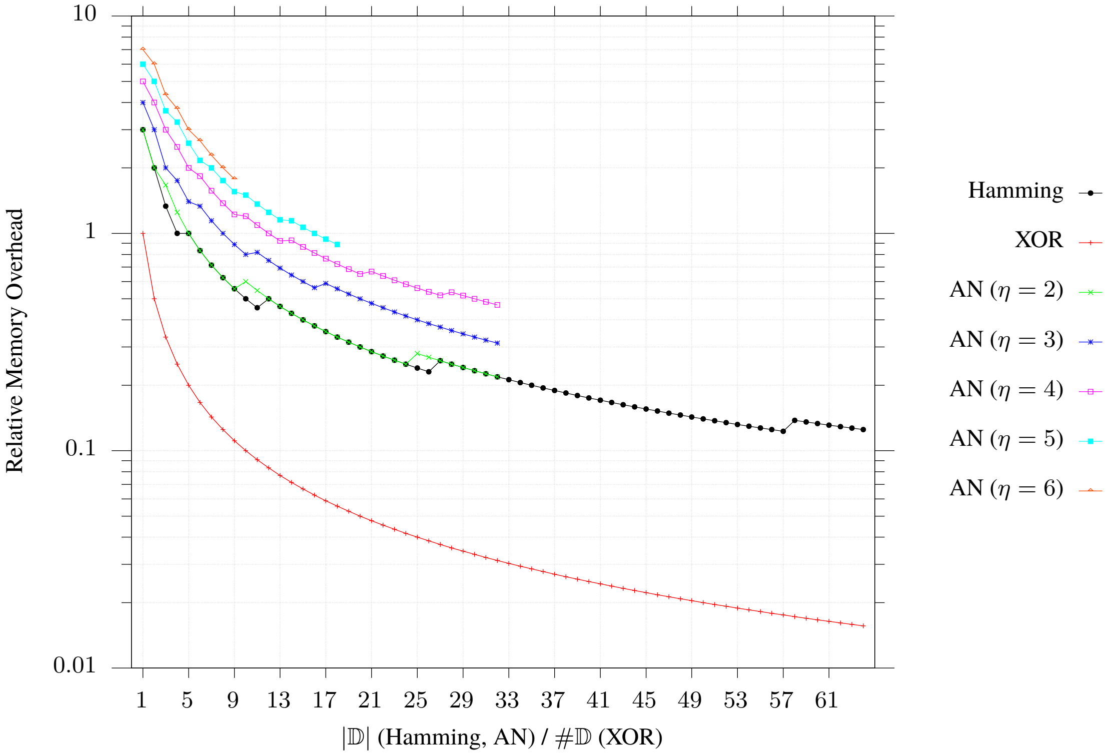
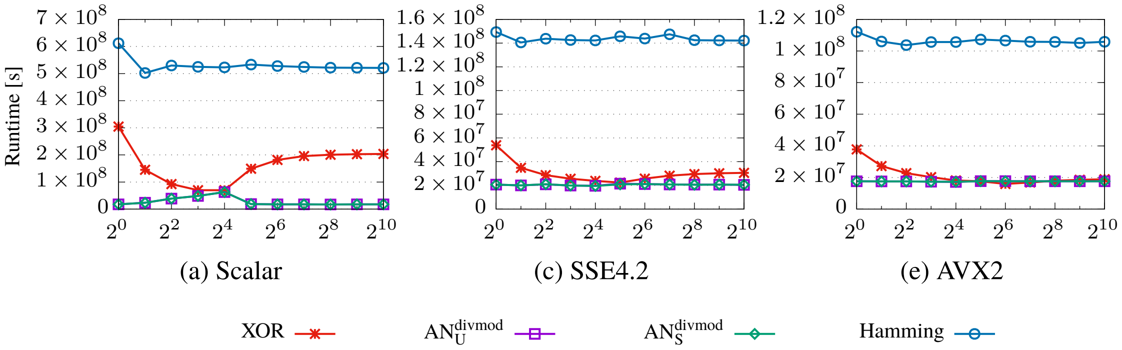


Figure 3.25: Memory overhead for (shortened) Extended Hamming and AN codes.

1. Hardening, 2. Error Detection, 3. Decoding, 4. Arithmetic, and 5. Aggregation . For each code and operations, we also measure runtimes for Scalar, SSE4.2, and AVX2 coding variants. As we described above, checksums are typically used for larger blocks of data. On the one hand, in the benchmarks we honor that fact and also test XOR checksum aspects for varying data block sizes. This allows certain optimizations like loop unrolling and much less data to be written. On the other hand, since Hamming and AN work on value granularity anyways, we will investigate *loop unrolling* for these two. This may allow some degree of optimization to compensate the inherent advantage of checksums (less data to be written in total). For Checksums, the block size indicates over how many *values* (not bytes) a checksum is computed, which are 16 bits wide for the sequential case, 128 bits for SSE4.2 and 256 bits for AVX2. So e.g. for SSE4.2 on 16-bit data, a block of size 64 contains $64 \cdot {}^{128\text{ bits per vector}}/_{16\text{ bits per value}} = 64 \cdot 8 = 512$ values. For AN and Hamming, the loop unrolling is designed the same, respectively. By that, when increasing the block size for XOR, less checksums are computed in total. For XOR, the unroll factor is the same as the block size. We rely on C++ template-meta programming to let the compiler unroll the code, for all codes. The SIMD algorithms (SSE and AVX) take into account that the number of values may not be aligned to the block size or loop unroll factor. That means, the scalar loop is used for the remaining values which do not fit into SIMD registers. We provide absolute runtimes for a total of one Billion 16-bit data values (100 001 integers × 10 000 iterations) per measured variant. Evaluation system 1 supports both SSE4.2 and AVX2, so that we can test them all, and we prohibit implicit vectorization with the GCC compiler flag `-fno-tree-vectorize`. Finally, for AN coding we used a 16-bit A resulting in 32-bit code words and we tested both signed and unsigned coding variants. We will see that there are sometimes significant differences in the runtimes. We will only concentrate on the direct comparison between the codes and only shortly elaborate in how far vectorization improves or worsens the operations' runtimes. For the sake of brevity, in the following comparisons, we will typically assume for each code the *best unroll factor* with the lowest runtimes and thereby the highest MIPS, if not otherwise noted. Several graphs are quite similar to each other, so here we



(a) Scalar

(c) SSE4.2

(e) AVX2

Figure 3.26: Hardening runtimes.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	8	2	256	256
SSE4.2	32	2	16	16
AVX2	64	4	16	16

Table 3.12: Unroll factors and block sizes with fastest hardening runtimes.

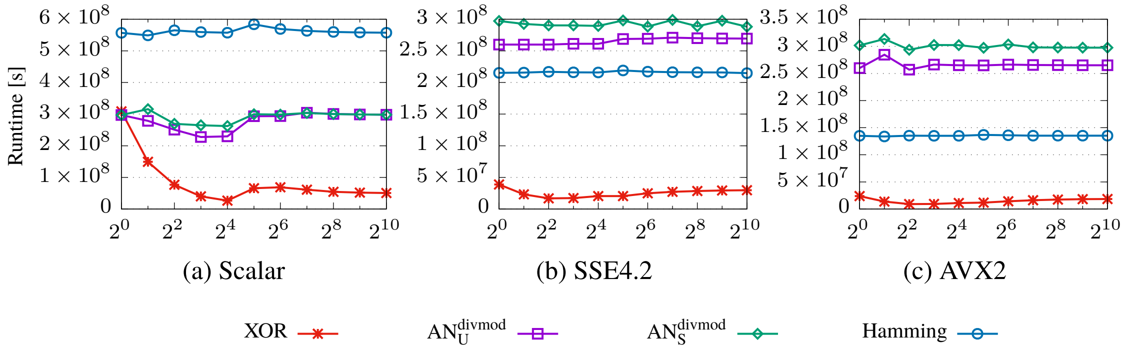
will only show a subset. The figures show *runtimes*, so that *lower values are better* and the y-axis scales often differ between scalar, SSE, and AVX measurement. Furthermore, here we only show results for the Skylake system and

Hardening

Hardening speed is important when e.g. (batch) loading (much) data into the database, or when new values are generated, or when values are recoded (see Section 3.4.3 below). Figure 3.26 displays results for hardening microbenchmarks. As the graphs suggest, software Hamming coding is almost an order of magnitude slower than XOR ($\approx 6 \ldots 7\times$) and AN ($\approx 6 \ldots 30\times$). XOR, in turn, is $4\times$ slower than AN for scalar execution (even up to $11\times$), but both are virtually equally fast for vectorized executions. Remember that the numbers represent the fastest runtimes per coding variant. Especially for hardening, we see no difference between unsigned or signed AN coding. XOR and Hamming benefit quite well from vectorizing, while AN runtimes worsen as it requires to convert the data types to wider ones and therefore needs additional instructions. Table 3.12 presents the best unroll factors for hardening runtimes, where we can see that mostly smaller unroll factors are sufficient to achieve very good results. Although scalar AN execution has its best runtimes with very high unroll factors, a factor of 1 gets close enough. For XOR and Hamming we can conclude that unroll factors between 2 and 16 achieve (close to) the best runtimes.

Error Detection

The next elementary operation is error detection, where all data is simply checked for bit flips, shown in Figure 3.27. For scalar execution (a), Hamming is $20\times$ as slow as XOR and $2\times$ as slow as AN. For vectorized execution, Hamming has runtimes which are $12.7 \ldots 14.8\times$ as high as for XOR. Compared to AN, Hamming is slightly faster for SSE ($1.2 \ldots 1.3\times$), but twice as fast for AVX. Comparing XOR and AN, the former is about an order of magnitude faster than the latter for scalar, SSE, and AVX execution ($8.5 \ldots 32\times$). For AN coding, we see that the unsigned modulus is executed slightly faster than the signed one, which will be seen in the later comparisons which include error detection, too. Again, XOR and Hamming benefit much better from vectorization than AN, since for them error detection is the same as hardening, while AN requires the much costlier modulus. Table 3.13 presents the best unroll factors for the error detection runtimes, where we can see that mostly smaller unroll factors are sufficient to achieve very good results. Although the SSE variants of Hamming signed AN have best unroll factors of 1024, we can conclude that factors between 2 and 16 achieve (close to) best runtimes all code variants.



(a) Scalar (b) SSE4.2 (c) AVX2

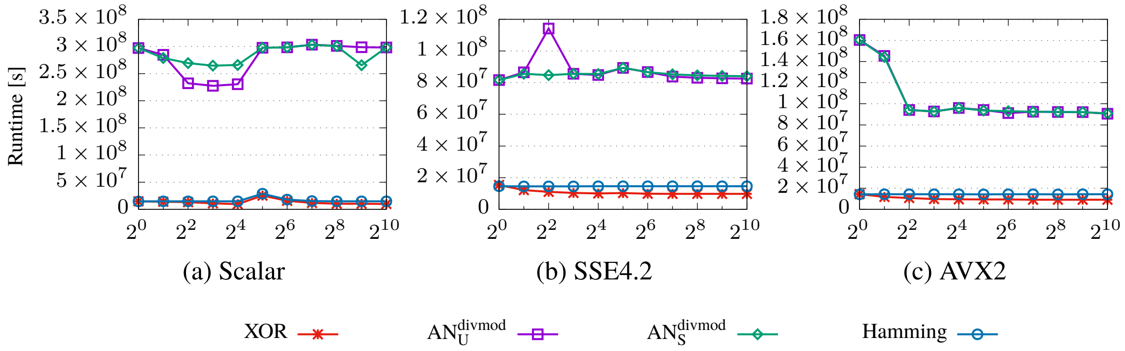
Figure 3.27: Error detection runtimes.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	16	2	8	16
SSE4.2	4	1024	4	1024
AVX2	4	2	4	4

Table 3.13: Unroll factors and block sizes with fastest error detection runtimes.

Decoding

Decoding may be required when the database has to work on unencoded data, or when shipping unencoded data to clients. Although decoding does not necessarily include the checking step, this would normally be the case – to be sure that we decode only valid data. We measure decoding with and without error detection to show the additional impact. Pure decoding for Hamming and Checksums is to copy the data without the code bits, which is trivial since both are systematic

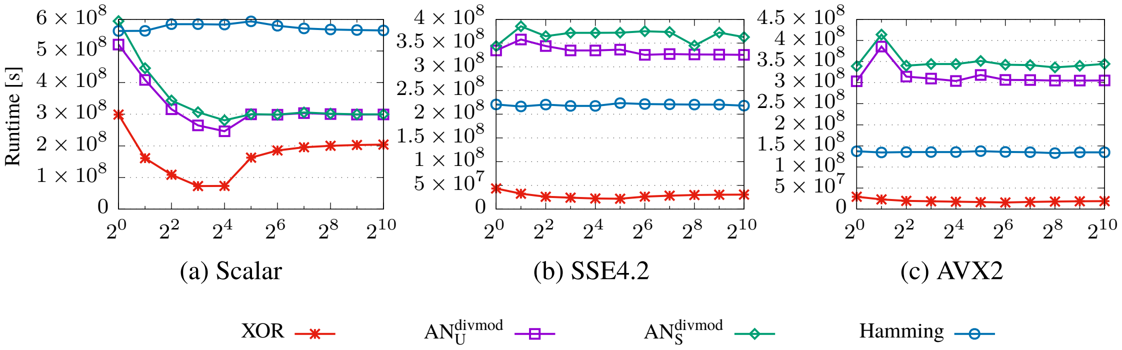


(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.28: Pure decoding runtimes.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	16	1	8	8
SSE4.2	256	4	1	1
AVX2	256	256	1024	1024

Table 3.14: Unroll factors and block sizes with fastest decoding runtimes.



(a) Scalar (b) SSE4.2 (c) AVX2

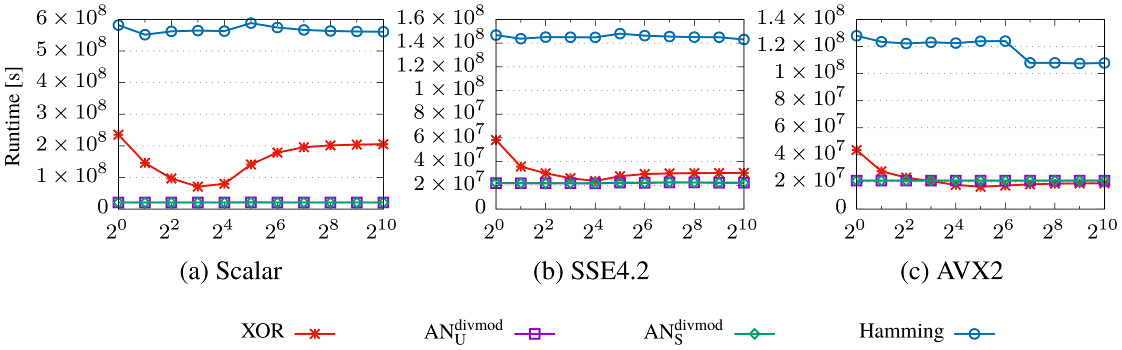
Figure 3.29: Decoding runtimes including error detection.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	8	1	16	16
SSE4.2	32	2	64	1
AVX2	64	256	1	256

Table 3.15: Unroll factors and block sizes with fastest decode-and-detect runtimes.

codes. In contrast, for AN coding we have to compute the unencoded value, i.e. divide. Figure 3.28 shows runtimes *without* detection. Here, AN is much worse than the other two coding schemes and we see that the scalar runtime is similar to that of the scalar error detection runtime. This means, that the CPU can execute the modulus about as fast as the division. For vectorized AN, we convert the integers to the next larger floating point numbers, as described in Section 3.3.4. For these there exist division operations and this is why for decoding, AN can benefit from the vectorization. However, AN is about an order of magnitude slower than XOR or Hamming, with runtimes being between $6 \ldots 28\times$ as high as for the other two codes. For Hamming and XOR, only the redundant code bits need to be "ignored" and the data bits can just be copied, which is why both seem about equally fast. XOR becomes faster than Hamming for larger block sizes, because there are less and less checksums which need to be skipped. Since decoding is a mere copying for XOR and Hamming, these differ only slightly, with XOR only about 34% faster than Hamming on average (for the fastest block sizes and unroll factors). In contrast to the previous operations, XOR and Hamming decoding (without detection) do not benefit from vectorization since they simply write out data, while AN slightly improves. Table 3.14 presents the best unroll factors for the decoding runtimes which are mostly in the lower range between 1 and 16. While some very high unroll factors achieve best runtimes for XOR (SSE and AVX) and AN (only AVX2), there again low unroll factors also achieve very good runtimes.

Figure 3.29 shows that, when enabling error detection before decoding, the detection primitives dominate the runtimes and the picture is quite similar to the pure detection runtimes from Figure 3.27, with Hamming overtaking AN coding when using vectorized execution. The graphs also show the difference in unsigned and signed runtimes for AN coding, which come to light even more so for vectorized execution. Since the runtimes are dominated by the detection part, Hamming and XOR again improve by vectorization in contrast to AN. Table 3.15 presents the best unroll factors for decode-and-detect runtimes, where the same picture as before is repeated: there are a few high unroll factors, but the majority is between 1 and 16. As the flat curves in Figures 3.28b and 3.28c indicate, low unroll factors achieve very good results there, too.



(a) Scalar

(b) SSE4.2

(c) AVX2

Figure 3.30: Pure addition runtimes.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	8	2	16	8
SSE4.2	16	1024	16	16
AVX2	32	512	512	16

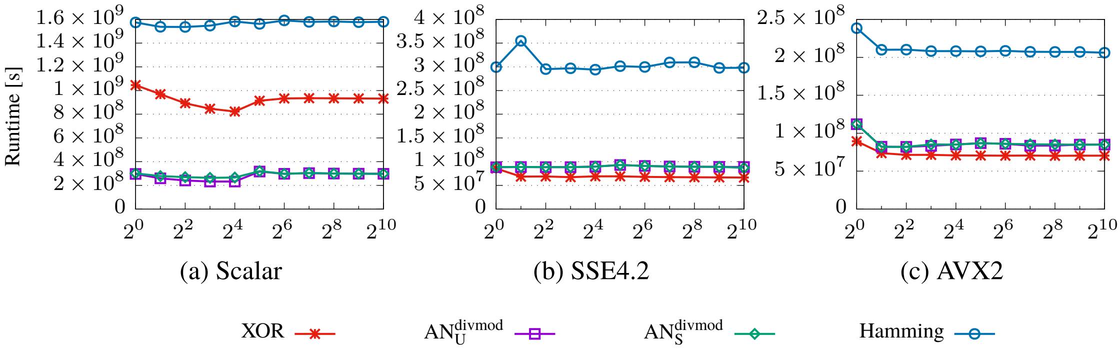
Table 3.16: Unroll factors and block sizes with fastest addition runtimes.

Arithmetic

We now consider arithmetic operations addition, subtraction, multiplication, and division, and show the runtime graphs for addition and division. These read the input values and write the result back into a result array. The graphs are similar for the first three, whereas division shows quite different behavior. The data is generated in a way such that no over- and underflows happen by not setting some of the most significant bits (MSBs) and LSBs. We first show the runtimes *without* error detection in Figures 3.30 and 3.31 and then *with* error detection in Figures 3.32 and 3.33.

Without error detection, AN coding can take full advantage of its arithmetic code properties: for addition, subtraction, and multiplication, it is more than 26×, 6× and 5× as fast as Hamming for scalar, SSE, and AVX execution, respectively. This is, because the Hamming codes must be recomputed for every value. Since division is much more costly, AN is then only about 6×, 3×, and 2× as fast as Hamming, respectively. Compared to XOR, AN is only faster for scalar execution, being slightly more than 3× as fast as XOR. However, when using vectorization, XOR can get as fast as AN for SSE and even slightly faster (1.2×) for AVX execution. For division, XOR is also 1.3× as fast as AN for SSE execution. XOR is 8× as fast as Hamming for scalar execution and 6× as fast for vectorized execution, while for division XOR is 2×, 4×, and 3× as fast for scalar, SSE, and AVX, respectively. For addition, subtraction and multiplication, XOR and Hamming again improve by a factor of $\approx 3 \dots 5$ when exploiting vector capabilities, whereas AN does not benefit for these three operations. For division, however, all codes benefit by factors of 5.2 and 7.5 for Hamming, 12.3 and 11.7 for XOR, and 2.6 and 2.8 for AN for SSE and AVX, respectively. For XOR, the AVX2 execution slightly degrades the runtime improvement over scalar, compared to SSE. Table 3.16 again shows the best unroll factors, with most of them being in the lower end. One exception is Hamming with AVX2, which has quite a big drop in runtime for high unroll factors, but this is still so much slower than XOR and AN so that its effect is diminishing.

The runtimes for the combination of arithmetic operations and error detection are shown in Figure 3.32 for addition, subtraction, and multiplication, which are again quite similar to each other, while division also exhibits quite different behavior as before, which is shown in Figure 3.33. XOR is about an order of magnitude faster than Hamming, with its best runtimes being 8.2×,



(a) Scalar

(b) SSE4.2

(c) AVX2

Figure 3.31: Pure division runtimes.

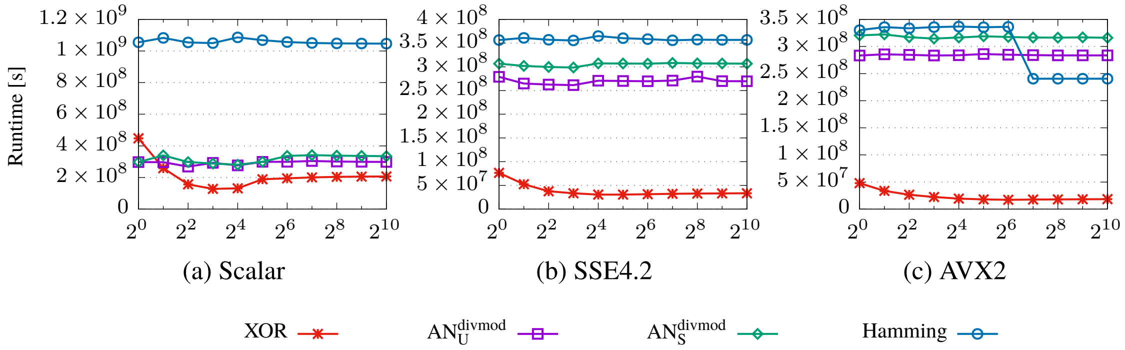
	XOR	Hamming	AN Unsigned	AN Signed
Scalar	16	4	8	8
SSE4.2	1024	16	1	1024
AVX2	256	1024	4	4

Table 3.17: Unroll factors and block sizes with fastest division runtimes.

≈ 12×, and ≈ 14× lower for scalar, SSE, and AVX, respectively. XOR is slower than AN coding for the scalar case, with best runtimes being ≈ 2× as high as for AN. However, the better vectorization potentials for XOR become clear again and it is ≈ 9× and ≈ 16× faster than unsigned and ≈ 10× and ≈ 18× faster than signed AN coding for SSE and AVX, respectively. Compared to AN coding, Hamming has ≈ 4× and ≈ 1.4 higher runtimes than AN for scalar and SSE, but for AVX2 Hamming even becomes slightly faster with runtimes being ≈ 1.3× lower than those of AN, with the unsigned AN variant being slightly faster then the signed one. Regarding the vectorization, Hamming improves by 3× and 4.4×, XOR by 4.2× and 7.4× for SSE and AVX2, respectively, while for AN the runtimes slightly worsen. As for the runtimes without error detection, all codes benefit much more from vectorization for division, especially where Hamming and XOR additionally benefit for the detection part. XOR improves by 4.4× and 6.6×, XOR by 13× and 12.6×, and AN by 1.6× and 1.5× for SSE and AVX2, respectively. Again we see that SSE provides even slightly better improvements than AVX2 does. Table 3.17 shows the same trend as before with low unroll factors between 1 and 16 making up the majority and Figures 3.31a to 3.31c indicate that for the cases with very high best unroll factors, low ones also perform very well. Hamming again exhibits the exception with AVX2, with quite a big drop in runtime for high unroll factors, but in this case Hamming becomes faster or as fast as than AN, for addition and division respectively.

Aggregation

Aggregation is a frequent database operation, too, with the main difference that typically many more values are read than are written back as result, contrary to the previous arithmetic operations. We measured computing the sum, average, minimum, and maximum over all values, so that only a single result value is written to the result array. However, since the sum can become quite large for so many values, the next larger native integer width is used, i.e. 32 bits for 16-bit data. The data is generated in a way such that this data width suffices to store the complete sum over all values by not setting some of the MSBs. All aggregation operation behave quite similar, so that in Figure 3.34 we only show the results for pure summation and in Figure 3.35 the runtimes for summation including error detection.

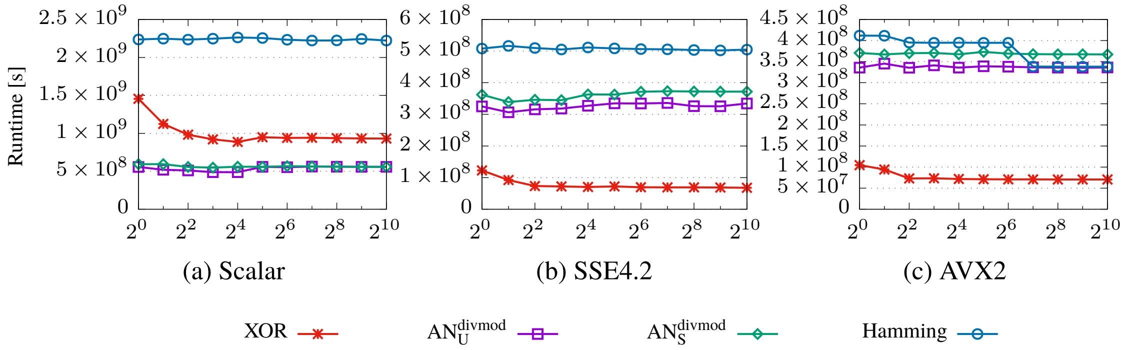


(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.32: Addition runtimes including error detection.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	8	1024	4	16
SSE4.2	16	8	8	8
AVX2	64	1024	1	8

Table 3.18: Unroll factors and block sizes with fastest add-and-detect runtimes.



(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.33: Division runtimes including error detection.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	16	128	8	8
SSE4.2	1024	512	2	2
AVX2	256	1024	4	2

Table 3.19: Unroll factors and block sizes with fastest divide-and-detect runtimes.

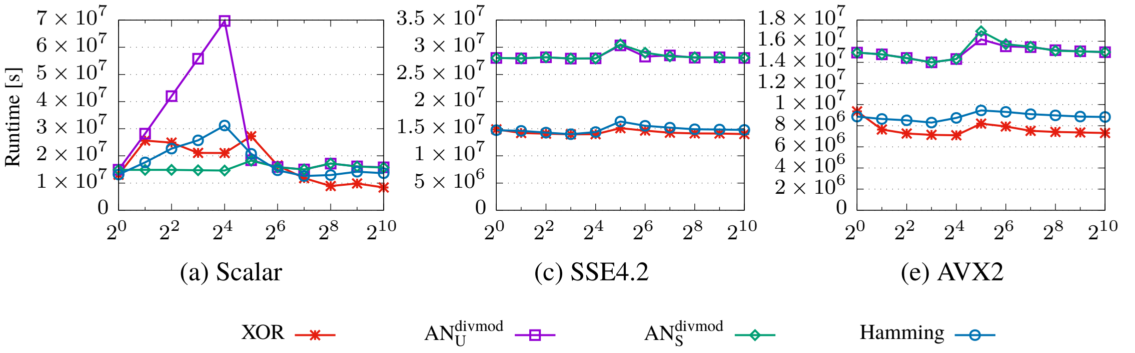
Without error detection, the scalar runtimes show some weird behavior for which we did not further investigate the cause, as the vectorized variants show much smoother behavior. XOR is slightly faster than Hamming for scalar execution by a factor of $1.5\times$, while with SSE both are equally fast and AVX2 brings a slight advantage for XOR again by a factor of $1.2\times$. Figure 3.34a shows however that this is achieved by XOR only for large block sizes and XOR is otherwise in the same range as Hamming. AN is about as fast as Hamming for scalar code with the fastest runtimes differing by a factor of $\approx 1.2\times$. As Table 3.20 shows, the best unroll factors are again more in the low end. Interestingly however, all codes get slower in the SSE case, due to the additional conversion to the larger data types and because then only half as many values can be summed up per vector instruction. With AVX2 this effect is reduced and the runtimes are improved over the scalar case on average by a factor of $\approx 2\times$.

Including error detection primitives in the aggregation operation again vaults the runtimes to the ranges from pure error detection. For scalar execution, Hamming is 70× slower than XOR for sum and average, and even 100× slower for computing minimum or maximum. For SSE and AVX the differences are still high with $15 \ldots 17\times$ for sum and average and $22 \ldots 24\times$ for minimum and maximum determination. Compared to AN, XOR is 30× and 20× faster for sum and average, and almost 50× and 30× faster for the latter two operations. AN is faster than Hamming only for the scalar case with a factor of $\approx 2\times$, but Hamming gets faster by $1.2 \ldots 1.4\times$ and $2.0 \ldots 2.3\times$ compared to unsigned and signed AN coding and SSE and AVX2, respectively.

Results for 32-bit data on System 1

We now summarize the results on system 1 for 32-bit data. We will first consider the effects of vectorizing the code, i.e. how much this can improve runtimes compared to scalar execution, and then compare the codings against each other using the actual runtimes. We will mostly relate all numbers and ratios to the findings from 16-bit base data.

For the basic coding operations hardening, error detection, decoding, and decode-and-check, the improvements through vectorization are much smaller for all codes and AN coding often even performs much worse with vectorization on 32-bit base data, compared to scalar execution. For the arithmetic operations the picture is quite the same: increasing the base data width to 32 bits makes the ratio between scalar and vectorized code worse than the ratio for 16-bit base data. While XOR and Hamming can still benefit from vectorization in these cases, AN coding does not benefit and for multiplication and division, the vectorized execution is even slower than the scalar one. When also doing error detection before the arithmetic operations, XOR and Hamming again have smaller improvements than for the 16-bit case, but SSE and AVX execution are still faster than scalar execution. For AN coding, however, the runtimes stay about the same and where for 16-bit base data only division could improve over scalar execution, with 32-bit base data this is not the case anymore. In contrast, for the aggregation operations the vectorization improvements are larger than for the 16-bit case for all codes. When error detection is also performed, vectorization again brings less gains than for the 16-bit base data case. Here, while XOR becomes slightly faster using SSE and AVX2, Hamming gets even slower and for AN vectorization has little to no effects.



(a) Scalar

(c) SSE4.2

(e) AVX2

Figure 3.34: Runtimes for computing the sum of all values.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	1024	128	1	16
SSE4.2	8	8	8	8
AVX2	16	8	8	8

Table 3.20: Unroll factors and block sizes with fastest summation runtimes.

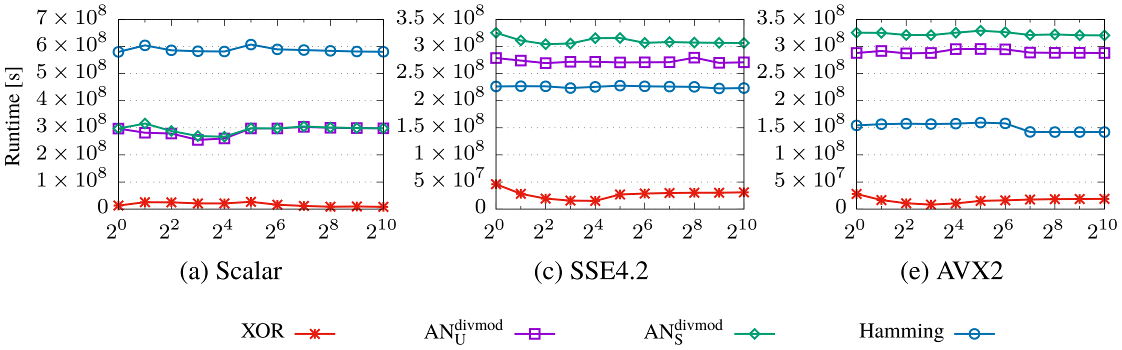


Figure 3.35: Runtimes for computing the sum of all values, including error detection.

	XOR	Hamming	AN Unsigned	AN Signed
Scalar	1024	1	8	16
SSE4.2	16	512	4	4
AVX2	8	1024	4	1024

Table 3.21: Unroll factors and block sizes with fastest sum-and-detect runtimes.

In total, the vectorization improvements are much smaller when increasing the base data width from 16 to 32 bits, and vectorization is mostly only useful for XOR and Hamming. Especially for aggregation, only AVX could really improve the runtimes over scalar execution.

Now, we consider the actual runtimes and compare the codes against each other. The first observation is that the runtimes are (sometimes substantially) higher than for 16-bit data. XOR and Hamming have virtually the same ratios as for 16-bit base data, but since XOR can benefit slightly better from vectorization than Hamming, the SSE and AVX runtimes differ even more in favor of XOR for 32-bit base data. This holds for all measured operations. Otherwise, AN coding becomes slower relative to Hamming coding compared to the 16-bit data case. It is still faster for hardening, whereas only 15×, 3.6×, and 2.9× for scalar, SSE, and AVX, respectively, while these factors were about 30×, 7.2×, and 6× for 16-bit data. The trend continues for the other operations, as well, so that AN coding is now typically slower than Hamming for all three execution variants. For arithmetic operations, AN is still faster than Hamming, but with typically smaller ratios. For error detection and detect-and-decode, AN is even slightly slower than Hamming, and now for the aggregation operations sum and average AN is even faster than Hamming, but for minimum and maximum determination AN is now slower than Hamming. Finally, while AN was faster than Hamming for aggregation and error detection for 16-bit data, it is now the other way around for all four operations on 32-bit data.

3.4.3 Runtime Adaptability

Requirement $\mathcal{R}3$ dictates that our error detection solution should be *adaptable* to changing error models at run-time. Hamming codes can hardly be adapted to changing error models due to its static structure. Data values would have to be split up into multiple smaller blocks, so that each data value is represented by a series of code words. This, however, requires *multiple* detection and decoding steps to access the whole value. XOR can be tailored to changing error models by decreasing the block width, or by increasing the checksum width. For the former, more checksums are inserted in between the data elements and for the latter, wider checksums must be recomputed using a coarser granularity. For AN codes we simply have to recode data with another golden

A. We are only considering this aspect on the coding level and it is implementation-specific how recoding must be realized in a concrete system.

3.5 PERFORMANCE OPTIMIZATIONS FOR AN CODING

Requirement $\mathcal{R}2$ (efficiency) demands that the approach introduces as little overhead as necessary. In the previous section, we showed that the memory overhead can be adjusted according to the data width and the required error detection capabilities to satisfy the memory aspect of Requirement $\mathcal{R}2$. Unfortunately, error detection and softening are based on division and modulo computations, which are expensive even on modern server-grade processors (cf. Figure 3.28 and [50]). For instance, on an Intel® Skylake processor as in our measurement system 1, Fog measured that a scalar division can take between 23 and 95 core clock cycles, while e.g. a scalar multiplication can take between 3 to 4 core clock cycles [50, p. 233]. The modulus operation is even more costly than the division, as it may lead to multiple division steps. We now show how to circumvent the costly division and modulus calculations and to the best of our knowledge we are the first to propose this for AN coding. Despite the fact that AN coding already received quite some attention, the following mathematical solution to faster AN coding has not been published so far [32, 41, 49, 51, 55, 69, 154, 155, 161].

3.5.1 The Modular Multiplicative Inverse

Today's Processors' ALUs work on the two's complement representation of integers. By that, they implicitly do computations in a residue class ring (RCR) modulo the power of two to the native register width, i.e. $\equiv 2^n$ $(n \in \mathbb{N})$ with typically $n \in \{3, 4, 5, 6\}$ which corresponds to 8, 16, 32, and 64 bits – today's CPUs' native register widths. In such an RCR, the division by any *odd* A can be replaced by multiplication with the *modular multiplicative inverse* A^{-1}:

$$A \cdot A^{-1} \equiv 1 \qquad (\mathrm{mod}\ M) \qquad , M \in \{2^8, 2^{16}, \ldots\} \tag{3.61}$$

In the following, we will refer to the modular multiplicative inverse only as *inverse* for short. Only odd numbers are coprime to any 2^n $(n \in \mathbb{N})$, so only for these there exists an inverse. Consequently, we restrict ourselves in the following to use *only odd* As. The determination of the *inverse* can not be done automatically by the compiler since the actual A and $\mathbb{D}_\Theta$ and thus $\mathbb{C}^A_{\mathbb{D}_\Theta}$ will only be known at run-time, as will be discussed later. Additionally, the code width may differ from the native register widths.

3.5.2 Faster Softening

The inverse allows to simplify the softening, which becomes:

$$c/A \equiv c \cdot A^{-1} \equiv (d \cdot A) \cdot A^{-1} \equiv d \qquad (\mathrm{mod}\ 2^{|\mathbb{C}^A_{\mathbb{D}_\Theta}|}). \tag{3.62}$$

The inverse can be computed with the Extended Euclidean algorithm (see e.g. [183, chapter10]) for the RCR modulo $2^{|\mathbb{C}^A_{\mathbb{D}_\Theta}|}$. When working with code word widths different from the processor's native register widths, the result must be masked, i.e. AND-ed with a bit mask having the appropriate $|\mathbb{C}^{A}_{\mathbb{D}_\Theta}|$ LSBs set to one. This is because there may be *ones* in the remaining MSBs from the multiplication. However, we can also compute the inverse for register widths so that we need not do any additional AND-ing operations. By that, we temporarily increase the code width, but not the original data width. Using the inverse has two very important advantages:

1. Using the inverse relieves Equation (3.18) from the division.
2. The inverse enables more efficient recoding from one code word $c_1 = d \cdot A_1$ into another $c_2 = d \cdot A_2$, by multiplying with the factor $A^* = A_1^{-1} \cdot A_2$:

   $$c_1 \cdot A^* = (d \cdot A_1) \cdot (A_1^{-1} \cdot A_2) = d \cdot A_2 = c_2 \tag{3.63}$$

   The product $(A_1^{-1} \cdot A_2)$ is a *constant factor* and needs to be calculated only once, especially when recoding multiple values, which we will use later.

The following example illustrates a code's properties:

Example 2. *For our example code from Example 1, where $A = 233$, and $|\mathbb{D}_\Theta| = 8$, it follows that*

$$\begin{array}{rrcrl} \textit{width of A} & |A| & = & 8 & \\ \textit{code width} & |\mathbb{C}^{A}_{\mathbb{D}_\Theta}| & = & 16 & \\ \textit{inverse} & A^{-1} & = & 55,129 & \textit{(unsigned)} \\ & & = & -10,407 & \textit{(signed)} \end{array}$$

3.5.3 Faster Error Detection

Using the multiplicative inverse allows to get rid of the modulo operator for error detection, too. For that, as in [69], we must know the largest and smallest encodable data word:

$$d_{\text{max}} = \max(\mathbb{D}_\Theta), \tag{3.64}$$

$$d_{\text{min}} = \min(\mathbb{D}_\Theta), \tag{3.65}$$

where the latter is required for signed integers, only. The following table summarizes the typical values for signed and unsigned integers:

	d_{min}	d_{max}
unsigned	—	$2^{\lvert\mathbb{D}_\Theta\rvert} - 1$
signed	$-2^{\lvert\mathbb{D}_\Theta\rvert-1}$	$2^{\lvert\mathbb{D}_\Theta\rvert-1} - 1$

Computations on code words must be done in the code word domain and thus on register widths $\geq |\mathbb{C}^{A}_{\mathbb{D}_\Theta}|$. Consequently, it follows that:

$$|c \cdot A^{-1}| = |c| = |d^*| > |d| = |c| - |A| \quad , d^* = d. \tag{3.66}$$

I.e., when softening, the resulting data word d^* is computed in the larger RCR (modulo $2^{|\mathbb{C}^{A}_{\mathbb{D}_\Theta}|}$) than the original data d requires (modulo $2^{|\mathbb{D}_\Theta|}$). This becomes very useful, because we discovered the *anomaly* that in this case it holds that:

$$d^* > d_{\text{max}} \to d^* = (d \oplus \varepsilon) \tag{3.67}$$

$$d^* < d_{\text{min}} \to d^* = (d \oplus \varepsilon) \tag{3.68}$$

where ε is an arbitrary, *detectable* bit flip of any weight and d^* was softened from a corrupted code word. This goes further than in [69], because we found that the anomaly occurs for *any* detectable error. For signed integers, the binary representation contains ones in the $|A|$ MSBs where there should be zeros after the multiplication with the inverse. Likewise, the same holds for negative integers, but now there are *zeros* in the $|A|$ MSBs, where there should only be ones. For unsigned integers, the first test suffices, while for signed integers both tests must be conducted. Consider the following example for signed integers (sint):

Example 3. *For our running example code from Examples 1 and 2 we set* Θ =signed integer (sint)*, so that* $|\mathbb{D}_{sint}| = |A| = 8 \Rightarrow |\mathbb{C}^{A}_{\mathbb{D}_{sint}}| = 16$*,* $A = 233$*, and* $A^{-1} = 55\,129$ *(unsigned)* $= -10\,407$ *(signed), and an example data word d* $= 5$*, multiplying with the inverse has the following effects:*

$$\begin{array}{rrllll}
1) & 5 \cdot A = 1165 & = & \overbrace{\ldots 0000}^{\textit{overflow}} & \overbrace{0000\,0100}^{|A|} & \overbrace{1000\,1101_2}^{|\mathbb{D}_{sint}|} \\
2) & 1165 \cdot A^{-1} & = & \ldots 0100 & 0000\,0000 & 0000\,0101_2 \\
3) & +1\textit{:}\ 1166 \cdot A^{-1} & = & \ldots 0100 & 1101\,0111 & 0101\,1110_2 \\
4) & -1\textit{:}\ 1164 \cdot A^{-1} & = & \ldots 0011 & 0010\,1000 & 1010\,1100_2
\end{array}$$

The first line shows the hardening, with the result in binary representation. The overflow column contains bits that may convolve out of the code width and must be masked when the used ALU registers are larger than $|A| + |\mathbb{D}_\Theta$. *The second line is the softening of the valid code word, with no zeros in the* $|A|$ *MSBs. Lines 3 and 4 show softening of altered code words, where* $1165 + 1$ *and* $1165 - 1$ *represent a double and single bit flip in the LSBs, respectively. For the signed case, line 3 triggers the test from Equation* (3.68) *and line 4 the one from Equation* (3.67)*. Here, the anomaly is visible in the* $|A|$ *MSBs.*

It is in principle possible to compute in RCRs other than modulus powers of 2. However, there may be rings where there exist no inverses for golden As [183, chapter10] and the effects we use for faster softening may not be applicable in other RCRs. Consequently, we assume only computing in RCRs modulo powers of 2, e.g. 8, 16, 32, or 64.

3.5.4 Comparison to Original AN Coding

Due to our coding improvements all AN coding operations can be naively vectorized, like for the other codes. Now, only multiplication and comparison operations are needed and supported by all considered SIMD ISAs. For shorter codes, especially $\leq$ 8-bit code words, not all instruction sets provide the integer `mullo` multiplication, where only the lower half of the actual result is stored. For these cases, special handling is required. For instance, up to SSE4.2 and AVX2 only 16-and 32-bit `mullo` is supported and AVX-512 adds 64-bit `mullo` [74]. Anyways, since AN coding is non-systematic, the code words are a single unit and we need not care about location of the code

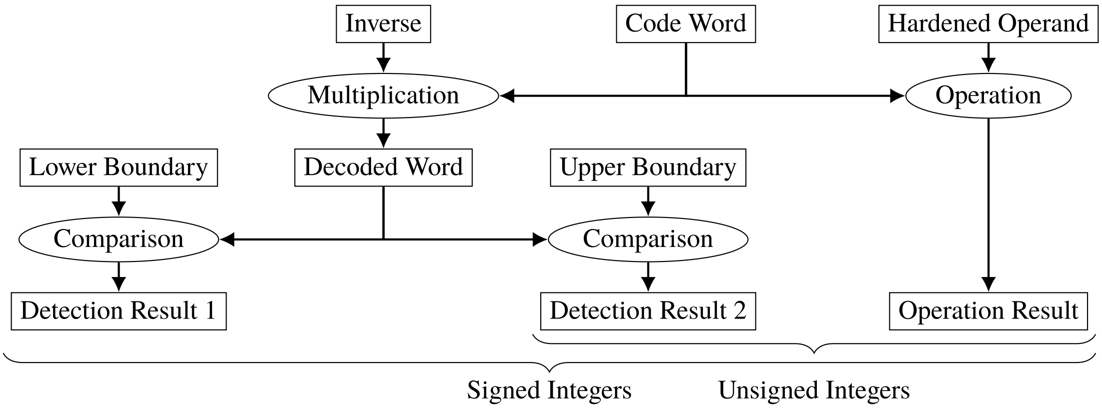


Figure 3.36: Parallelism in AN coding operations using the multiplicative inverse.

bits. By that, we need less complex algorithms than for Hamming. Since data processing and modification can be applied directly to the code words, no special handling of code bits is required. Additionally, softening (multiplication with the inverse) and data processing or modification can be applied in parallel by multiple vector units. Figure 3.36 visualizes the concept, where rectangles represent values and ellipses denote operations. The *multiplication* of code word and inverse leads to an intermediate (decoded word), which is then *compared* according to Equations (3.67) and (3.68) against the data type's lower and upper bounds. Meanwhile, the actual *operation* can be performed on the original code word and the hardened operand while there are no data dependencies. When dealing with signed integers, we need two *comparison*s and have up to three parallel data flows which can be efficiently dealt with by out-of-order CPUs, while signed integers require only one comparison and have up to two parallel data flows.

To confirm the improvements for coding and processing throughput, we conduct the same microbenchmarks as in Section 3.4.2. Since the improvements only have an impact on error detection and decoding, we will only conduct those benchmarks again which include the error detection primitives. The operations without error detection have virtually the same runtimes as before. In the following, we again only compare the fastest runtimes and we show only a subset of the available graphs. As can be seen from Figures 3.37 to 3.41, AN coding now is on a par with XOR, except for a very few cases. For instance, for decoding, vectorization *degrades* the runtimes compared to scalar code (Figures 3.38b and 3.40c), and for some cases AN is even faster than XOR, e.g. for scalar detect-and-decode (Figure 3.39a) or scalar detect-and-add (Figure 3.40a). We will now compare the runtimes of only the original and the improved AN coding. The reader can see the difference in the graphs below, with the original AN coding shown as dotted lines and the improved one with solid, decorated lines. Pure error detection (Figure 3.37) improves by factors of $6.1\times$, $10\times$, and $18.8\times$ for the unsigned case, and by factors of $6.1\times$, $6.5\times$, and $13.1\times$ for the signed case. Pure decoding (Figure 3.38) looks quite different, where the unsigned and signed variants improve for scalar execution by factors of $15.6\times$ and $18.1\times$, respectively, and for SSE by $2.6\times$ and for AVX2 by $4.6\times$. While pure decoding does not improve as much as detection for the vectorized cases, when performing error detection prior to decoding, the picture is again more like for pure error detection (Figure 3.39). Then, for the unsigned case, the improved AN coding is faster than the original by factors of $5.2\times$ (scalar), $7.9\times$ (SSE), and $12.1\times$ (AVX), while the signed variant improves by factors of $5\times$ (scalar), $6.4\times$ (SSE), and $10.9\times$ (AVX). This shows that the costly modulo operation for error detection had by far the greater impact on performance than the division for softening. For the simpler arithmetic operations addition and subtraction (Figure 3.40), the improvement factors are for unsigned $4.1\times$ (scalar), $9.1\times$ (SSE), and $12.9\times$ (AVX), while for the signed case they are $4\times$ (scalar), $6.4\times$ (SSE), and $12.8\times$ (AVX). For multiplication the factors are slightly different with $3.6\times$ (scalar), $7.7\times$ (SSE), and $12.5\times$ for unsigned integers and $3.7\times$ (scalar), $5.8\times$ (SSE), and $11.4\times$ (AVX) for signed integers. As division is costly anyways, the same factors are much lower with $1.8\times$, $3\times$, and $3.8\times$ for unsigned, and $1.8\times$, $2.9\times$, and $3.9\times$ for signed integers. For the aggregation operations (Figure 3.41) those factors are for unsigned and signed integers $3.8\ldots5\times$ (scalar), $6\ldots8.9\times$ (SSE), and $11.7\ldots18.4\times$ (AVX), where the variants for signed integers are the slower ones due to the second comparison which is required here.

The above shows that softening and error detection become much faster with the improved AN coding. By that, AN coding is now fast enough to be considered for employing in main memory-centric DMSs. As a consequence, recoding of hardened data now seems viable, too. It is only a multiplication with a *constant factor*, as was shown in Equation (3.63), which in turn is as fast as hardening. Given this fact, we need to adjust our Definitions 5 and 6 of data hardening and softening to accommodate this ability.

Definition 11 (Data Hardening, Revised). *We denote the process of* increasing *the bit flip detection capability by either* encoding *unencoded data or by using some error code-specific means as* ***data hardening****. For AN coding, this means using a* larger *A with better detection guarantees, or for XOR, we would use a larger checksum width. The term "encoding" will henceforth only be used if and only if an* unencoded *value is hardened.*

Definition 12 (Data Softening, Revised). *We denote the process of* reducing *the bit flip detection capability as* ***data softening****. For AN coding, this translates into using a* smaller *A with worse detection guarantees. For XOR, this would mean using a smaller checksum width. The term "decoding" will henceforth only be used if and only if a* hardened *value is returned to its unencoded form.*

3.5.5 The Multiplicative Inverse Anomaly

We want to provide a more formal explanation of the modular multiplicative inverse *anomaly*. Therefore, we will now first provide an illustrative explanation of why multiplying invalid AN code words with the multiplicative inverse will lead to bits being set or not set in the $|A|$ MSBs of the code word, where it should be the other way around. Afterwards, we provide two inequalities that must hold for bit flips to be detectable. We cannot give a complete formal proof since the convolution of multiplication is a hard problem, which is due to the unpredictability of carries[11].

We first have to recapitulate that decoding is done in the code word domain, so that

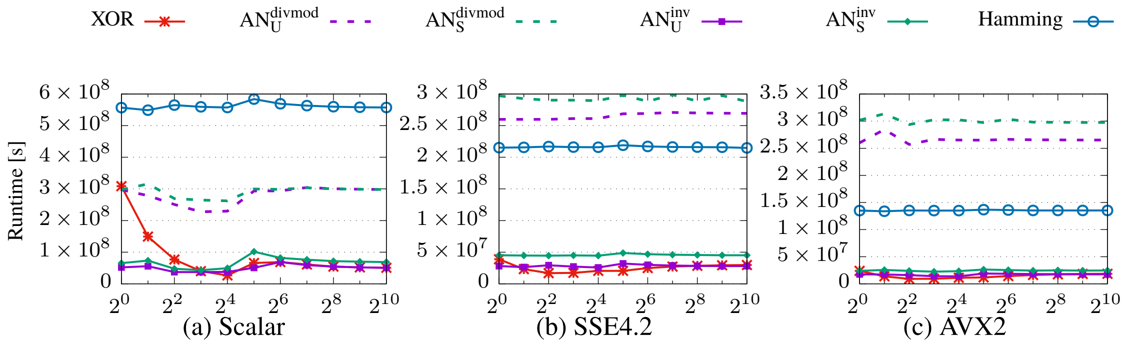
$$c \cdot A^{-1} = d^* \qquad , |c| = |A^{-1}| = |d^*| < |d|$$

and we can write a data word d as the sum of its bits d_i:

$$d = \sum_{i=1}^{x} (d_i \cdot 2^{i-1}) \qquad , x = |d|, \, d_i \in \{0, 1\}.$$

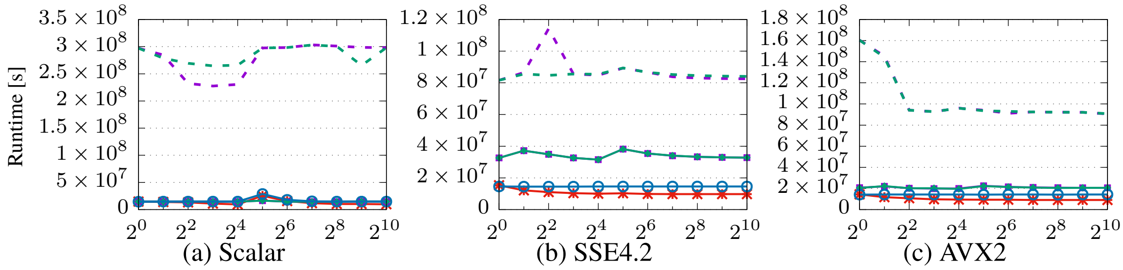
In coding theory, this means we are computing in the *Galois Field $GF(2)$* where words are composed of only ones and zeros. The same holds of course for any code word c and its bits c_i,

[11] This is in contrast to, e.g., the binary XOR operation which is binary addition without carries



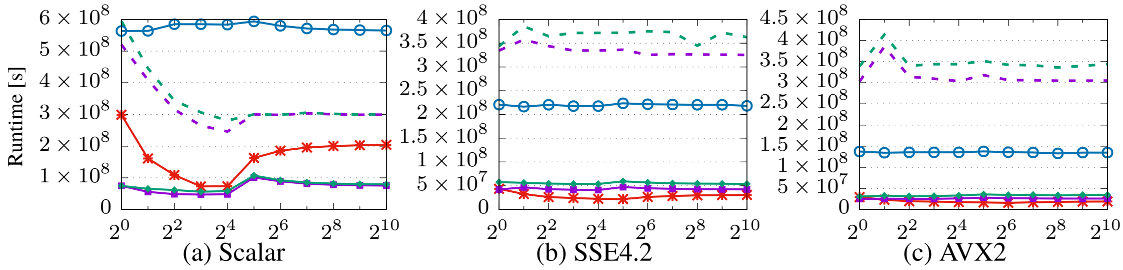
(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.37: Error detection runtimes for original and improved AN (Skylake).



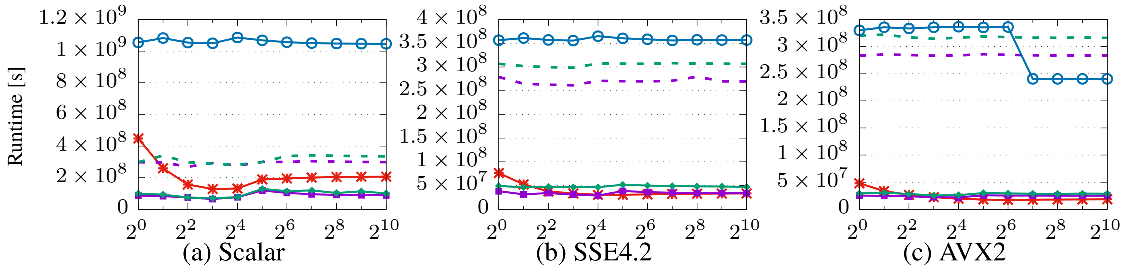
(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.38: Decoding runtimes of original and improved AN (Skylake).



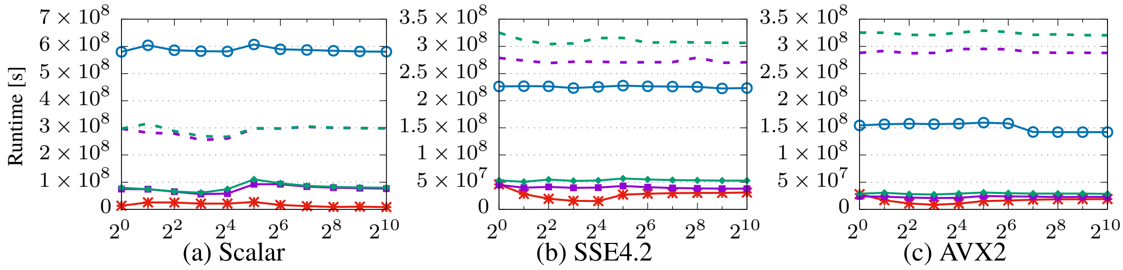
(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.39: Detect-and-decode for original and improved AN (Skylake).



(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.40: Detect-and-add for original and improved AN (Skylake).



(a) Scalar (b) SSE4.2 (c) AVX2

Figure 3.41: Detect-and-sum of original and improved AN (Skylake).

XOR
AN_U^{divmod}
AN_S^{divmod}
AN_U^{inv}
AN_S^{inv}
Hamming

respectively. Using that notation we can express the decoding as:

$$d^* = c \cdot A^{-1} = \sum_{i=1}^{x}\sum_{k=1}^{x}(c_k a_i^{-1} \cdot 2^{i+k-2}) \qquad , (c_k, a_i^{-1}) \in \{0,1\} \tag{3.69}$$

$$= d \cdot A \cdot A^{-1} = \sum_{i=1}^{x}\sum_{k=1}^{x}(d_k a_k a_i^{-1} \cdot 2^{i+k-2}) \qquad , (d_k, a_k, a_i^{-1}) \in \{0,1\} \tag{3.70}$$

The problem at hand is very hard due to the convolution, i.e. all the carries being unpredictable. The following illustrates Equation (3.69) used for decoding of some AN code for which an inverse actually exists and where the previous requirements hold that $d \in \mathbb{D}_\Theta$, $c \in \mathbb{C}^A_{\mathbb{D}_\Theta}$, and $x = |c| = |A^{-1}| = |d^*| > |d|$:

$$\begin{array}{c|ccccccccc}
 & & & c & & \cdot & & A^{-1} & & \\
= & c_x & c_{x-1} & \ldots & c_1 & \cdot & a_x & a_{x-1} & \ldots & a_1 \\
\cline{2-10}
= & & & & & & c_1a_x & c_1a_{x-1} & \ldots & c_1a_1 \\
\oplus & & & & & & & & \vdots & \\
\oplus & & & & & c_na_x & c_na_{x-1} & \ldots & c_na_1 & \\
\oplus & & & & & & & \vdots & & \\
\oplus & & & & c_xa_x & c_xa_{x-1} & \ldots & c_xa_1 & & \\
\cline{2-10}
= & d^*_x & d^*_{x-1} & & & \ldots & & & & d^*_1 \\
= & & & & & d^* & & & &
\end{array}$$

Above, each code word element is multiplied with each element of the inverse and all products are added together. For valid code words, in the resulting d^* the $|A|$ MSBs must either all be zero (for unsigned or positive integers) or all be ones (for negative signed integers). Consequently, for a *valid* decoded unsigned word it holds that

$$d^* = 2^{x-1}d^*_x + 2^{x-2}d^*_{x-1} + \ldots + 2^{|\mathbb{D}_\Theta|}d^*_{|\mathbb{D}_\Theta|+1} + 2^{|\mathbb{D}_\Theta|-1}d^*_{|\mathbb{D}_\Theta|} + \ldots + 2^0 d^*_1 \tag{3.71}$$

$$0 = d^*_x = d^*_{x-1} = \ldots = d^*_{|\mathbb{D}_\Theta|+1} \tag{3.72}$$

$$\Rightarrow d^* = 2^{|\mathbb{D}_\Theta|-1}d^*_{|\mathbb{D}_\Theta|} + \ldots + 2^0 d^*_1 \tag{3.73}$$

For signed integers, this holds appropriately, whereas Equation (3.72) equals zero for positive and equals one for negative signed integers, respectively.

Now, imagine a single bit flip that causes one of the code word elements to flip from one to zero, e.g. at position n. Then, the multiplication will miss the addend, e.g. starting with c_na_x (indicated as red line), transforming d^* into a erroneous word d^*_ε:

$$\begin{array}{c|ccccccccc}
 & & & c & & \cdot & & A^{-1} & & \\
= & c_x & c_{x-1} & \ldots & c_1 & \cdot & a_x & a_{x-1} & \ldots & a_1 \\
\cline{2-10}
= & & & & & & c_1a_x & c_1a_{x-1} & \ldots & c_1a_1 \\
\oplus & & & & & & & & \vdots & \\
\cline{6-9}
\oplus & & & & & & & \vdots & & \\
\oplus & & & & c_xa_x & c_xa_{x-1} & \ldots & c_xa_1 & & \\
\cline{2-10}
= & d^*_x & d^*_{x-1} & & & \ldots & & & & d^*_1 \\
= & & & & & d^*_\varepsilon & & & &
\end{array}$$

Since we only consider *invalid* code words, $c \neq 0$ must hold, so there *must* be some ones which are not propagated out of the non-data positions. From that it follows, that there must be some leftover ones in the $|A|$ MSBs which are *not* carried out. Conceptually, it is the same when a zero flips to a one, where one of the lines which was formerly plain zero now has some value.

We will now develop inequalities which describe when bit flips are detectable. Therefore, we model a bit flip not only as an XOR operation, but as the addition of some error number ε:

$$\begin{aligned} c &= d \cdot A \\ d^* &= c \cdot A^{-1} = d \cdot A \cdot A^{-1} = d \cdot 1 = d \\ c_\varepsilon &= c + \varepsilon = d \cdot A + \varepsilon && (3.74) \\ d_\varepsilon^* &= c_\varepsilon \cdot A^{-1} = (d \cdot A + \varepsilon) \cdot A^{-1} = (d \cdot A \cdot A^{-1}) + (\varepsilon \cdot A^{-1}) \\ &= d + \varepsilon \cdot A^{-1} && (3.75) \end{aligned}$$

Since $\varepsilon \neq 0$, it must hold that $A^{-1} \cdot \varepsilon \neq 0$, as well.

Example 4. *Example for the anomaly, tailored to Equations* (3.74) *and* (3.75) *in decimal and binary formats. For readability, the binary representation is separated into groups of 4 bits.*

1)	*5 (d)*	$\cdot$	*233 (A)*	=	*1,165 (c)*	*mod* 2^{16}
	0000 0000 0000 0001_2		0000 0000 1110 1101_2		0000 0100 1000 1101_2	
2)	*1,165 (c)*	$\oplus$	*15*	=	*1,154 (c_ε)*	
	0000 0100 1000 1101_2		0000 0000 0000 1111_2		0000 0100 1000 0010_2	
3)	*1,165 (c)*	+	*65,525 (ε)*	=	*1,154 (c_ε)*	*mod* 2^{16}
	0000 0100 1000 1101_2		1111 1111 1111 0101_2		0000 0100 1000 0010_2	
4)	*1,165 (c)*	$\cdot$	*55,129 (A^{-1})*	=	*5 (d)*	*mod* 2^{16}
	0000 0100 1000 1101_2		1101 0111 0101 1001_2		0000 0000 0000 0101_2	
5)	*1,154 (c_ε)*	$\cdot$	*55,129 (A^{-1})*	=	*48,946 (d_ε^*)*	*mod* 2^{16}
	0000 0100 1000 0010_2		1101 0111 0101 1001_2		1011 1111 0011 0010_2	
6)	*65,525 (ε)*	$\cdot$	*55,129 (A^{-1})*	=	*48,941*	*mod* 2^{16}
	0000 0100 1000 0010_2		1101 0111 0101 1001_2		1011 1111 0010 1101_2	

Example 4 shows concrete examples for Equations (3.74) and (3.75). There, lines 2 and 3 are equivalent regarding the 4-bit flip resulting in the erroneous code word c_ε. This shows that we can model a bit flip, which is equivalent to an XOR operation, as an integer addition, too. Lines 4–6 show that $c_\varepsilon \cdot A^{-1} = d + \varepsilon \cdot A^{-1}$. While line 4 decodes the valid code word, the result of line 5 shows the anomaly in the upper $|A| = 8$ bits. Line 6 resembles Equation (3.75):

$$d_\varepsilon^* = d + \varepsilon \cdot A^{-1} \Leftrightarrow \varepsilon \cdot A^{-1} = d_\varepsilon^* - d.$$

With the numbers from Example 4 we obtain

$$\overbrace{65,625}^{\varepsilon} \cdot \overbrace{55,129}^{A^{-1}} = 48,941 = \overbrace{48,946}^{d_\varepsilon^*} - \overbrace{5}^{d}.$$

Algorithm 3.2 Verify Equations (3.67) and (3.68)

```
1:  for $A \in \{3, 5, 7, \ldots, (2^{16} - 1)\}$ do
2:      for $|\mathbb{D}| \in \{1, 2, \ldots, 24\}$ do
3:          $|\mathbb{C}| \leftarrow |A| + |\mathbb{D}|$
4:          $A^{-1} \leftarrow euclidean(A, |\mathbb{C}|)$        ▷ euclidean algorithm for arbitrary code word widths
5:          $d_{\text{min}} \leftarrow -2^{|\mathbb{D}|-1}$
6:          $d_{\text{max}} \leftarrow 2^{|\mathbb{D}|-1} - 1$
7:          $c_{\text{min}} \leftarrow A \times d_{\text{min}}$
8:          $c_{\text{max}} \leftarrow A \times d_{\text{max}}$
9:          for $c = c_{\text{min}}, c_{\text{min}} + 1, \ldots, c_{\text{max}}\ ,\ c \not\equiv 0 \pmod{A}$ do        ▷ Test all non-code words
10:             if $d_{\text{min}} \leq (c \times A^{-1}) \leq d_{\text{max}}$ then
11:                 Indicate Error
12:             end if
13:         end for
14:     end for
15: end for
```

We now combine Equations (3.67), (3.68) and (3.75) by setting $d^* = d^*_\varepsilon$. It follows that a bit flip ε, which is representable as the addition $c_\varepsilon = c + \varepsilon$, is *detectable* as soon the following two *inequalities* are fulfilled

$$d + \varepsilon \cdot A^{-1} > d_{\text{max}} \quad \Rightarrow \quad \text{detectable bit flip} \tag{3.76}$$

$$d + \varepsilon \cdot A^{-1} < d_{\text{min}} \quad \Rightarrow \quad \text{detectable bit flip} \tag{3.77}$$

where for unsigned integers only the first inequation applies. Furthermore, Equations (3.76) and (3.77) show that even when ε and thus c_ε are multiples of A, the bit flip is detectable as soon as the inequations are satisfied.

We confirmed the anomaly experimentally for all odd As with $2 \leq |A| \leq 16$, signed integers with $1 \leq |\mathbb{D}_{\text{sint}}| \leq 24$, and $|\mathbb{C}^A_{\mathbb{D}_{\text{sint}}}| = |A| + |\mathbb{D}_{\text{sint}}|$. Note that it is irrelevant whether we interpret integers as signed or unsigned, because the multiplication is the same for both in two's complement number representation and this is the standard for all modern processors. Algorithm 3.2 showcases that we validated Equations (3.67) and (3.68) using an exhaustive test over all possible code words. Lines 1 and 2 iterate over all odd As and all data widths up to 24 bits, respectively. From these we derive the code word width (line 3), which denotes the exponent for the RCR's power of two and is the input to our extended euclidean algorithm (line 4). From the data type (width) we can implicitly define d_{min}, d_{max}, c_{min}, and c_{max}, the largest and smallest values in the data type and code domain, respectively (lines 5–8). Finally, we test for all code words, whether their product with the inverse is outside the original data domain, as denoted in Equations (3.66) to (3.68). On a 64-core AMD Opteron 6274 server it took almost 50K CPU hours in total, about 780 hours per core or almost 33 days.

3.6 SUMMARY

There is a wide range of error control code families, which are typically designed for communication systems, where forward error correction is often desired to avoid retransmission of data packets. In

		SW-Hamming	XOR Checksum	Improved AN coding
	Fine-Grained Detection	✓	✗$^{(1)}$	✓
Qualitative Overhead	Encoding	high–medium	low	low
	Detection	high	low	low
	Decoding*	high	low	low
	Arithmetic*	high	low	low
	Aggregates*	high	low	low
	Memory Overhead	$\frac{\text{\# parity bits}}{\text{\# data bits}}$	$\frac{\text{\# checksum bits}}{\text{\# bits per block}}$	$\frac{\lvert A \rvert \text{ bits}}{\text{\# data bits}}$
	Runtime Adaptivity	✗$^{(2)}$	✗$^{(2)}$	✓
	Instruction Coverage	✗	✗	Arithmetic$^{(3)}$
	Error Detection	SEC-DED	all odd weights$^{(4)}$	$1\ldots^{(5)}$

Table 3.22: Summary of properties of the error detection techniques. For the overheads we can only do relative rating, as these depend on various factors. *: including error detection. $^{(1)}$: typically, checksums are computed over larger blocks of data. Single-value is possible but all overheads increase a lot. $^{(2)}$: adaptivity through different block sizes. $^{(3)}$: Errors in arithmetic operations are detectable, e.g. addition, but not in logical ones, e.g. comparisons. $^{(4)}$: depends on the actual checksum and is very complex to calculate. $^{(5)}$: Depends on data width and A, see Section 3.2.3 and e.g. Table 3.6.

the data management domain, however, there are other requirements, first and foremost effectiveness to detect a certain amount of bit flips in main memory (cf. Requirement $\mathcal{R}1$), and moreover the possibility to implement them in software in a way that they run efficiently (cf. Requirement $\mathcal{R}2$). We chose 3 code families, namely XOR checksums, software Hamming, and AN coding, where the former two are systematic, linear block codes, while the latter is a non-linear, non-systematic arithmetic code. The findings which we summarize in the following are condensed in Table 3.22.

After introducing their structure and basic coding operations for encoding, error detection, and decoding, we investigated their effectiveness in terms of the probabilities of silent data corruption (SDC). There, we revised from literature the properties for Hamming coding, but for XOR checksums there is virtually no literature available and for AN coding the existing results are inappropriate. In particular, since DMSs support many different data widths and want to use as few additional error code bits as necessary, we had to investigate the properties of XOR checksums and AN codes in much greater detail than was done before. For checksums, the error detection capabilities depend on the block length. One simple observation is, that all *even* numbers of bit flips in the same bit locations (referring to the final parity word) cancel each other out. We derived several rules to compute the actual conditional probabilities for arbitrary checksum (word) widths and arbitrary numbers of data words aver which the checksum is built. By that, we can iteratively compute all probabilities for 1-, 2-, and 3-bit checksums. For AN codes, the detection capabilities depend on both the data length and the parameter A. It is required to do brute force determination of the SDC probabilities, because they are non-linear codes. This is done by comparing all code words against each other, for which the effort grows exponentially with a factor of 4. To mitigate this, we showed that using grid approximation, which uses equidistant numbers, yields better (smaller) approximation errors than pseudo- or quasi-random numbers. Furthermore, we provide a freely available implementation for *GPU clusters*[12], and we provide a table for all data widths between $1, \ldots, 32$ bits and parameter widths between $2, \ldots, 16$ bits.

[12] See `https://brics-db.github.io/coding_reliability`.

Subsequently, we discussed using vector instructions of modern processors to accelerate the coding operations. This is extremely important, because modern in-memory DMSs extensively use all hardware features to improve system throughput and latency. As we showed, XOR operations are available in several vector ISAs (cf. Table 3.10), while for Hamming the population counting needed to compute the individual parity bits currently needs to be emulated. For the latter, we investigated and measured three different techniques, where we also showed that on our two measurement systems (cf Table 3.7), different techniques are fastest ones. For the original AN coding, only the encoding step, requiring multiplication, is supported by the vector ISAs.

We then conducted an extensive comparison of the three selected error codes. First, we compared their effectiveness regarding detection granularity, detection capabilities, and instruction coverage. Regarding the first, only Hamming and AN are designed to work on value granularity, while XOR checksums are better suited for larger blocks of data. Regarding the second aspect, we compared the detection capabilities. Regarding the third, only AN coding allows to detect errors in CPU instructions, because it is an arithmetic code. In contrast, for Hamming and XOR the code bits must be computed separately, which does not allow detecting bit flips in CPU instructions. Afterwards, we compared their efficiency, where we first analyzed their memory consumption. XOR and Hamming have fixed consumptions, which only depend on the number of data words and the data width, respectively. For AN this is more complicated, since it depends on both the data width and the chosen parameter. We could show that with those golden As which exhibit the same error detection capabilities as Hamming in terms of guaranteed minimal detectable bit flip weight, AN can have mostly the same overhead as Hamming. Afterwards, we conducted microbenchmarks to compare not only the speeds of the basic coding operations (encode, decode, and error detection), but also some arithmetic (addition, subtraction, multiplication, and division) and aggregation (sum, average, minimum, maximum) operations which are typical for data management and processing systems. Table 3.22 summarizes these as *qualitative overheads*.

The benchmarks showed that original AN coding has severe throughput issues, which we afterwards addressed by using the *modular multiplicative inverse*. This brings many benefits: First, costly division and modulo operations are replaced by a single multiplication for decoding, and additional comparisons for error detection. Second, coding then relies only on multiplication and comparison, which are supported by the considered vector ISAs. Third, by using the inverse, error detection consists of several independent data paths, which provides opportunities for out-of-order execution CPUs to schedule multiple independent instructions more efficiently (cf. Figure 3.36). These benefits are reflected in the speeds of coding operations, which we then again compared against the other codes. In essence, by using the inverse, operations can improve by a factor of up to $\geq 18\times$ for AVX execution. As a consequence, AN code operation speeds are close to those of the block-oriented XOR checksums, while Hamming coding is mostly much slower than XOR checksums and AN coding. Finally, as we showed, for AN coding the recoding of data is like encoding, a multiplication with a constant factor.

In this chapter, we fulfill Requirement $\mathcal{R}1$ – *effectiveness* – in the sense that we can choose for concrete error models, i.e. bit flip rates and weights, a desired error code which guarantees a desired minimal detectable bit flip weight. Due to the AN coding improvements, we fulfill Requirement $\mathcal{R}2$ – *efficiency* – for coding operations and we found two error codes where all code operations can be vectorized. We will investigate later in how far this impacts query runtimes. Furthermore, as we provide AN coding parameter tables for many combinations of data and parameter bit width, we can fulfill our Requirement $\mathcal{R}3$ – *adaptability* – in the sense that we can choose at runtime values from our computed parameter tables. Based on the above findings, we conclude that AN coding is suitable for value-granularity bit flip detection on integers, while XOR should be used for bit flip detection on larger blocks of data.

4

BIT FLIP DETECTING STORAGE

In this chapter, we use the two previously selected coding schemes, AN coding and XOR checksums, to build a hardened storage layer for in-memory DMSs. Column stores are the de-facto standard for in-memory data management systems. The earliest successfully introduced systems were MonetDB in the late 1990's [21, 23, 24], and C-Store in the early 2000's [4, 176]. Up to now, all major vendors released column store extensions to existing database systems or standalone products, e.g. SAP HANA [47], IBM DB2 BLU [147], Microsoft SQL Server Hekaton [40], or Oracle TimesTen [95]. However, the problem of unreliable hardware was so far mainly left to the hardware and operating system. To change this, we provide the following main contributions in this chapter:

1. We show that hardening in general is orthogonal to most if not all aspects of the storage of data.
2. We show that AN coding in particular can be combined with many of the integer data representation techniques.
3. We show that data hardening can thus be integrated in a holistic manner into the storage layer of data management systems.
4. We extend the ubiquitous B-Tree index data structure with a few different methods for detecting bit flips in the tree nodes, published in [88].

.

To the best of our knowledge, we are the first to integrate arbitrary bit flip detection in a holistic manner into the storage layer of main memory-centric column stores. In the following, we first revise the storage layer of modern column stores in Section 4.1. There, we discuss the basic storage architecture, which includes the logical data types (Section 4.1.1), the storage model (Section 4.1.2), the physical data representation (Section 4.1.3), physical data layouts (Section 4.1.4), as well as index structures (Section 4.1.5). We will use this to show that hardening combinable with most if not all of these aspects in Section 4.2. We describe how to harden data using the selected data coding schemes, AN coding and XOR (Section 4.2.1). Afterwards, we demonstrate that the used coding schemes are well combinable with lightweight data compression (Section 4.2.2). Then, we discuss restrictions regarding the physical data layout when introducing data hardening (Section 4.2.3). Finally, we also harden the ubiquitous B-Tree, which is a pointer-heavy data structure (Section 4.3). Section 4.4 summarizes and concludes the chapter.

4.1 COLUMN STORE ARCHITECTURE

DMSs were first and foremost developed to provide user applications an *abstraction* of how data is physically structured and stored [159]. In this section we will revise the storage layer of modern in-memory column stores. We will discuss the basic architecture, including logical data types in Section 4.1.1, physical storage models in Section 4.1.2, physical data representations in Section 4.1.3, physical data layouts in Section 4.1.4, and finally index structures in Section 4.1.5.

The text book architecture of DMSs is called ANSI/SPARC architecture and is illustrated in Figure 4.1. DMSs typically employ 3 *layers* of abstraction so that *client application*s need not care how to physically structure and store data, but rather work with a *logical* data model and

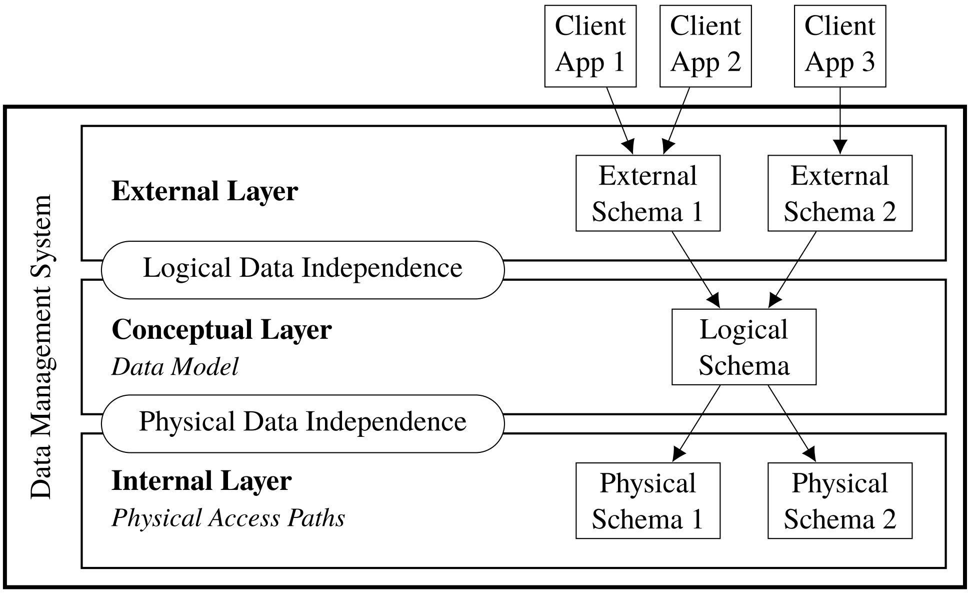


Figure 4.1: ANSI/SPARC three level architecture of DMSs, after [159]

declarative data query language. First, the result of data modeling, database design, and data definition is the *logical schema*. This schema is maintained on the *conceptual layer* and describes e.g. the tables (for relational databases). There can be multiple client applications which access the same database. In the *external layer*, for each such application there may exist an *external schema* containing (part of) the logical representation of the data to be stored, in form of e.g. views. The logical schema is uniform for the individual external schemas and decouples the applications from the *physical schemas*, which a DMS chooses to define the actual data representation, data layout, and physical access paths. The physical schemas are maintained in the DMS's *internal layer*. For relational DMSs, the logical schema is to some degree standardized through concepts in the structured query language (SQL)[1] standard, whereas each DMS vendor typically provides some slightly different variant. In contrast, the physical schema is completely specific to the individual DMS.

In the following, we are interested in the lower two layers. A more detailed view is given in Figure 4.2 which shows the concepts that each schema specifies. First, we will discuss the *logical* data types provided by the logical schema in the conceptual layer, which we regard in this thesis. Afterwards, we examine options for the *physical* storage model, data representation, and data layout, which all constitute the physical schema. The storage model defines the *primary* and *secondary* access paths, which means how to arrange the base data (tuples and attributes) and which index structures can be used. The data representation further refines the storage model by mapping the logical data types to underlying, implementation language specific, physical data types and data structures. This mapping is very specific to each DMS. This also includes potentially reducing the data domain by some data compression. Finally, the data layout again refines the data representation by defining how the actual bits and bytes are arranged in main memory. This lowest layer from the ANSI/SPARC architecture is of particular interest: it is the part which is changed by introducing data hardening coding into data management systems. Due to the decoupling of logical and physical schema, we only need to change this last layer and the client applications need not know about any error control coding done inside the DMS.

[1] The ISO/IEC JTC 1/SC 32 group defines many SQL and related standards, see `https://www.iso.org/committee/45342.html`.

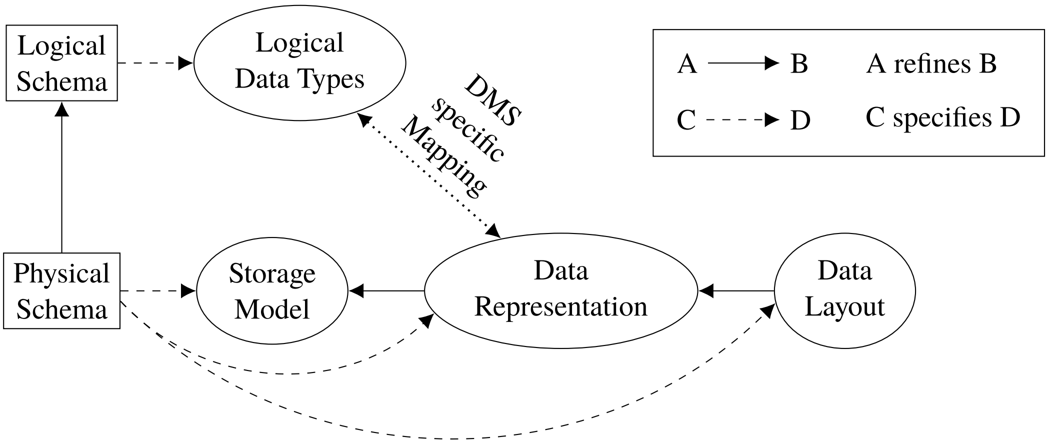


Figure 4.2: The correlation between the logical and physical schemas and the concepts they specify.

4.1.1 Logical Data Types

Column stores typically support a fixed set of basic, logical data types, including integers, decimal (fixed- or floating-point) numbers, and strings. Table 4.1 provides a running example for this chapter, visualizing a table with four columns (attributes). This example table represents summaries of orders, containing (1) an *ID* column as primary key attribute of type *integer*, (2) another integer-type column with the total number of items in the order, (3) the order's total prize represented as *decimal* number with 2 digits after the dot, (4) the shipping mode of that order as a *string*, and, finally, (5) the date and time when the order was submitted as a *timestamp*. The different standards for the SQL define varying amounts of logical data types. Some of the major types[2] used are categorized as follows: *Integer* data types such as TINYINT, SMALLINT, INT (or INTEGER), BIGINT, and HUGEINT. The BOOLEAN type for binary true/false information. *Character* data types like CHAR (or CHARACTER) for fixed-length strings, VARCHAR for variable-length strings, or TEXT, CLOB (Character Large Object), BLOB (Binary Large Object) for large, unbounded strings. For the remainder of this thesis, however, we do not consider large objects (BLOBS or CLOBS). There are *Decimal* data types like DECIMAL or NUMERICAL, and *floating-point* numbers like REAL, FLOAT, and DOUBLE. *Temporal* data types such as DATE, TIME, or TIMESTAMP, allow specifying either a date, a time, or both and optionally a time zone.

DMS users, like application developers, only deal with the logical data types from the conceptual level. The DMS internally maps them to physical data schemas, which consist of three parts: (1)

[2] We use the types given in `https://www.monetdb.org/Documentation/Manuals/SQLreference/BuiltinTypes`.

	ID	Number of Items	Total Order Prize	Shipping Mode	Order Date
	2348534	5	173 665.47	“Mail”	2017-10-25 14:58 CET
	2348535	10	46 929.18	“Air”	2017-11-04 13:37 CET
	2348536	3	193 846.25	“Rail”	2017-11-10 10:42 CET
Type	INTEGER	TINYINT	DECIMAL	VARCHAR	TIMESTAMP

Table 4.1: Logical Data Types in a Column Store

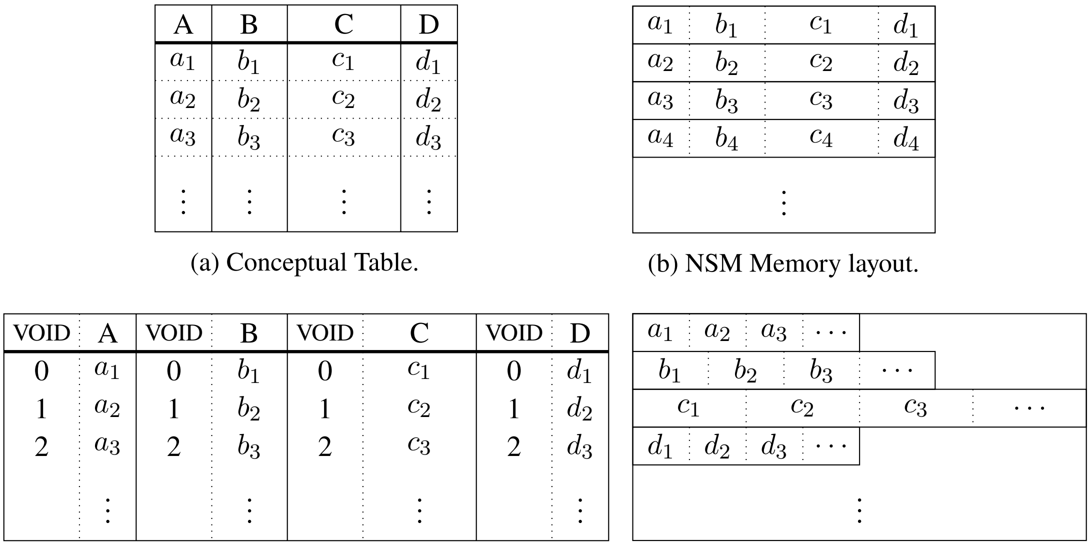
the *storage model*, (2) the *data representation*, and (3) the *data layout*. In this order, the parts have a decreasing degree of abstraction, where the storage model is a coarse description of the physical layout of tuples and their values. The storage model allows (or makes feasible) to use specific data representations, which are mappings from logical to physical data types. Finally, the data layout defines in which order bits and bytes are actually stored. We will now describe the parts in the given order.

4.1.2 Storage Model

In-memory column stores keep all business data and intermediate results for query processing exclusively in main memory. One major difference was the introduction of a new data storage model, which deviates significantly from classical row stores [40, 73, 96, 176, 200], as Figure 4.3 illustrates. This layout practically gives column stores their name and this layout is adopted by virtually all major main memory centric data management system. We will use the conceptual table from Figure 4.3a to clarify the differences, which contains 4 attributes A, B, C, and D, and a few rows representing individual tuples. The differing column widths represent different effective data widths of the columns' data types. For instance, the columns may match those from Table 4.1 in the same order. The classical n-ary storage model (NSM) [36] used in row-oriented DMSs stores all tuples in a row-wise fashion as shown in Figure 4.3b. There, all attributes of one tuple are stored consecutively and then the next tuple is appended likewise. In contrast, column stores maintain relational data using the decomposition storage model (DSM) [36], where each column of a table is separately stored as a fixed-width dense array [3], as illustrated in Figure 4.3d. This can be done either as a single large vector or as multiple smaller vectors, but that decision is irrelevant for our discussion. For decomposition storage model (DSM), to be able to reconstruct the tuples of a table, each column record is stored in the same (array) position across all columns of a table [3]. This index, also called object identifier (OID) [21], can be stored explicitly so that a column is stored as pairs of OID and value. OIDs can also be purely virtual, so that they are then called virtual object identifiers (VOIDs) and not stored physically, as shown in Figure 4.3c. For instance, since December 2016 MonetDB does not store any OID- or VOID-value-pairs, but employs a so-called *headless* mode where only either values or OIDs are stored. Ailamaki *et al.* introduced a hybrid storage model, called Partition Attribute Across (PAX), which tries to combine the best of NSM and DSM [7]. There, each database page is stored NSM-like, containing all of a table's attributes, but inside the page the attributes are organized in a DSM fashion. By that, each column is further divided into multiple vectors. The concept is very much as shown in Figure 4.3d, where the illustrated block would only be a single database page. PAX is a model variant that lies between the two extremes of NSM and DSM and since any considerations concerning DSM can also by applied to PAX, we do not further distinguish between these two and, without loss of generality (w.l.o.g), we only use the term of columnar storage.

4.1.3 Data Representation

The physical data representation is the second step in determining the physical schema. It is comprised of two parts: a mapping between logical and physical data types and, optionally, compression based on the chosen physical data type. The latter is of important, because DSM allows using lightweight compression schemes, which are otherwise less feasible with NSM.



A	B	C	D
a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_3	c_3	d_3
⋮	⋮	⋮	⋮

(a) Conceptual Table.

a_1	b_1	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_3	c_3	d_3
a_4	b_4	c_4	d_4
⋮			

(b) NSM Memory layout.

VOID	A	VOID	B	VOID	C	VOID	D
0	a_1	0	b_1	0	c_1	0	d_1
1	a_2	1	b_2	1	c_2	1	d_2
2	a_3	2	b_3	2	c_3	2	d_3
	⋮		⋮		⋮		⋮

(c) Conceptual table extended with VOIDs.

a_1	a_2	a_3	$\cdots$
b_1	b_2	b_3	$\cdots$
c_1	c_2	c_3	$\cdots$
d_1	d_2	d_3	$\cdots$
⋮			

(d) DSM Memory layout.

Figure 4.3: Row-wise NSM and column-wise DSM layouts.

Physical Data Types

The purpose of physical data types is to choose a data type provided by a certain programming language to provide a physical form for the logical types. In the following, we provide primitive type names known from the open-source C++ compiler GCC[3], which can easily be mapped to other languages' or compilers' data types.

For fixed-width data types, e.g. single character, integer, decimal and floating-point, column stores utilize basic arrays of the respective type for the values of a column [3, 73]. Some column stores like MonetDB utilize the smallest available native integer type for representing integer-based logical types [21]. Boncz and Kersten distinguish between logical, physical, and implementation types and the latter corresponds to what we here call physical data types. In [21], Boncz and Kersten use their physical data types as an indirection to be able to switch between actual implementation types for the same logical data type. This is really only used for their `OID` data type, which can be implemented as `VOID`, which requires no actual physical storage, or as a 4-byte wide integer. Apart from that, they store `TINYINT` as 1-byte integer (`uint8_t`), `SMALLINT` as 2-byte integer (`uint16_t`), `INT` as 4-byte integer (`uint32_t`), and `HUGEINT` as 8-byte integer (`uint64_t`).

Regarding decimal numbers, for the sake of correctness and accuracy, database systems typically use fixed-point numbers and arithmetic instead of native floating point numbers (`float` / `double`). Neumann provides detailed background information and discussion on this issue in a blog article [129]. In essence, rounding and precision loss problems of native floating-point numbers and operations are usually unacceptable for database systems. This is why production systems employ fixed-point arithmetic and there also exist several open-source libraries providing a variety of mathematical operators [52, 56, 153]. One possibility of representing fixed-point numbers is to split a number into powers of 100, so-called *limbs*, e.g. $1024 = 10 \cdot 100^1 + 24 \cdot 100^0$. In this case, each limb fits into a single byte and the position of the decimal point is stored separately, for instance in column metadata, but of course the limbs can also be larger. Interesting here is what we call the *limb domain abundance* (LDA) for such a representation, which means how much of

[3] `https://gcc.gnu.org/`

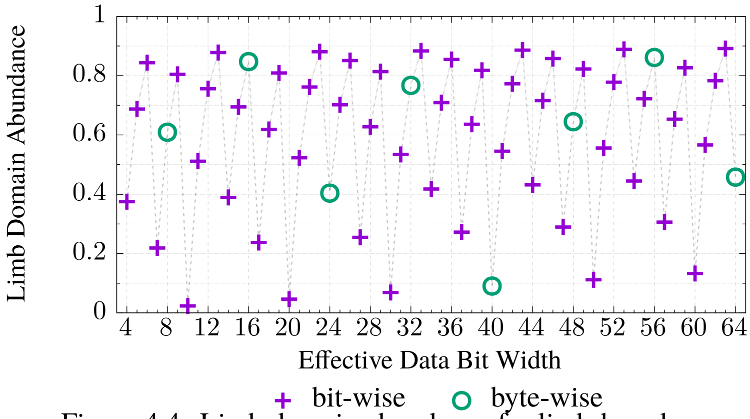


Figure 4.4: Limb domain abundance for limb-based representation of decimal numbers (lower is better).

n	LDA	n	LDA
8	0.609	10	0.023
16	0.847	20	0.046
24	0.404	30	0.069
32	0.767		
40	0.091	40	0.091
48	0.645	50	0.112
56	0.861	60	0.133
64	0.458		

Table 4.2: Selection of numerical values for the limb domain abundance (LDA) from Figure 4.4.

the *data domain* (number of representable values) stays *unused*. It is computed for a bit width of n by

$$\mathrm{LDA}(n) = 1 - \frac{10^{\lfloor \log_{10}(2^n) \rfloor}}{2^n}.$$

For instance, for single-byte limbs, 100 out of 256 numbers are represented per limb, which results in

$$\mathrm{LDA}(8) = 1 - \frac{100}{256} \approx 0.61,$$

of the representable data domain to stay unused. In contrast, when e.g. using 4-byte limbs, 10^9 out of ca. $4 * 10^9$ numbers are used per limb, resulting in a domain abundance of

$$\mathrm{LDA}(32) = 1 - \frac{10^9}{2^{32}} \approx 0.77.$$

As we regard powers of 10, 4 bits are the smallest limb size. Figure 4.4 depicts this data domain overhead for bit- and byte-widths up to 64 bits and we can see that the data domain efficiency is best for multiples of ten. In contrast, the byte-wise limb representations have mostly very high domain overheads, except for the 40-bit case. Table 4.2 shows a selection for bit- (right) and byte-wise (left) limb representations for direct comparison. However, the principal criterion is of course the range of decimal numbers that shall be represented. Based on the data distribution, a desired limb width can be chosen. On the one hand, it may be selected such that it minimizes the *total* memory consumption, by choosing the smallest limb width which can store “most” of the numbers in a single or small amount of limbs. On the other hand, it may be chosen such that it minimizes the *amount of work* for processing “most” of the numbers by using byte-wise limbs. For instance, for 64-bit limbs the LDA is “only” ≈ 0.46, which is better than for the other register-aligned limb widths of 8, 16, or 32 bits. Furthermore, 8-byte limbs can represent 19 orders of magnitude ($10^{\lfloor \log_{10} 2^{64} \rfloor} = 10^{19}$), or a range of ten quintillion numbers. The representation of decimal numbers dates back to the early 1950s, when White considered coding single-digit decimals. While he describes different methods to code the binary numbers to decimal digits, the assumption was that such techniques would be integrated into circuits. However, not all computer systems provide hardware support for decimals, but the Intel® x64 ISA provides a few instructions to work on binary-coded decimal (BCD) integers[4]. These are unsigned 4-bit integers storing a single decimal digit and up to 18 digits can be stored in a packed 80-bit format in x87 FPU data registers, i.e. a single 80-bit register

[4] See e.g. https://software.intel.com/en-us/articles/intel-sdm volume 1, or directly https://software.intel.com/sites/default/files/managed/a4/60/253665-sdm-vol-1.pdf. Visited 21 March 2018.

contains 18 4-bit limbs. According to the Intel® documentation, such a packed format can store numbers in the range of $-10^{18}+1\ldots10^{18}-1$. As we can see, we can achieve almost the same using a simple 64-bit representation with ranges $0\ldots10^{19}-1$ or $-10^{18}\ldots10^{18}-1$, but then have to deal with overflows, underflows, carries, etc. in software. As can be easily seen, the number of operations for dealing with these arithmetic issues increases with decreasing limb sizes or with increasing decimal ranges.

The temporal data types can be represented also as integers, e.g. as unix timestamps. The time zone can be fixed for a whole column, as converting between time zones is a rather trivial task.

For variable-width data types like *strings* it is more complex, because these can be physically represented in many different ways. A big problem is the large variety of character sets and character encodings. The former is a mapping from glyphs (letters, numbers, symbols, etc.) to numbers, while the latter is a mapping from the character set numbers to bit sequences, or directly from glyphs to bit sequences. Today, the most well-known character set is probably Unicode[5], which defines not only almost all characters of all of today's languages, but also many symbols and even emojis, which we all simply refer to as *glyphs*. Today, version 10 is the latest one, which supports 136,690 glyphs in total[6]. However, Unicode only defines the mapping between glyphs and numbers, and there are many different character encodings. The best known encoding today may be UTF-8[7], which encodes each Unicode glyph to a *sequence* of octets (single bytes) and fully incorporates the important US-ASCII [8] character encoding. UTF-16 is an older format, which encodes the Unicode glyphs using one or two 16-bit units. US-ASCII is an encoding which maps 127 glyphs (in its original form) to a byte representation. In Europe, the ISO/IEC 8859-1 standard, also known as *Latin-1*, is also often used and it also incorporates the whole US-ASCII encoding. The important difference between these encodings is the range of glyphs: US-ASCII and Latin-1 only support limited ranges in a way that all glyphs can be stored using a single byte. In contrast, using the UTF-8 or UTF-16 encoding, glyphs may be stored with *varying* amounts of bytes, at most up to 4 bytes. In practice, this becomes even more complex, as Unicode allows combined characters such as special accented characters, which actually requires multiple glyphs. In principle, this poses a similar problem as for decimal number representation. Both store the information in limbs of various sizes, whereas the size is typically fixed for one columns or vector. To offload this problem from query processing, for string columns nowadays some kind of dictionary coding is applied to transform them into fixed-width-data columns [1, 3, 18]. This was used in MonetDB already as enumeration type, so that low-cardinality columns could be represented with smaller physical data types [21]. As a consequence, columns of the logical string data type actually store integers which point into a separate data structure. We will cover more details in the following.

Lightweight Compression

Already a few decades ago, data compression was investigated for data management systems. At this time, the main benefit was to reduce the I/O costs in terms of throughput (less data to read) and seek times (less data accesses) of disk-based systems [76]. This directly translated into reduced I/O transfer times and increased buffer hit rates [58, 60, 76, 199]. Meanwhile, main memory systems were already considered as well by Roth and Van Horn [158]. For the latter, the aim is to store not only as much base data as possible, but also to compress the working set of transactions,

[5] `https://www.unicode.org/`

[6] `https://www.unicode.org/versions/Unicode10.0.0/`, visited 24 March 2018.

[7] `https://tools.ietf.org/html/rfc3629`

[8] United States-American Standard Code for Information Interchange

to avoid running out of main memory and enabling as many parallel in-memory transactions as possible. Since then, researchers considered both primary (columns) and secondary (indexes) data access paths [58, 186], as well as query optimization [34]. To store as much data in fast main memory as possible, especially light-weight compression plays a crucial role in modern in-memory data management systems [34] and particularly for column stores [145, 158]. Here, DSM shows another great advantage over NSM: compression of values of the same data type (DSM) allows for much better compression rates and speeds than compressing tuples with many different data types (NSM) [4]. Here, we consider lightweight compression techniques, because they play a crucial role for in-memory DMSs. Later, we will investigate whether and how AN coding can be combined with these techniques. However, we do not consider any string compression techniques in this thesis, but only *integer* lightweight compression.

Damme *et al.* distinguish between *techniques* and *algorithms*, where techniques work either on the logical or physical data representation, and an algorithm is the incarnation of one or (the combination of) more techniques [37]. Following that, one typically first applies one or more logic schemes to reduce the value domain, and then apply a physical technique – typically some form of null suppression – to actually reduce the required storage. Damme *et al.* identified five basic lightweight compression techniques: dictionary encoding (DICT), run length encoding (RLE), frame of reference (FOR), delta encoding (DELTA), and null suppression (NS). The compression techniques are separated into logical and physical ones [37]. All schemes except Null Suppression are considered *logical*, which means they typically aim to represent data as integers, or to reduce the numerical values that represent the stored information [37]. In contrast, Null Suppression is considered *physical*, which reduces the actual number of bits used to store data [37]. The “goodness” of a compression scheme heavily depends on the actual data and the actual workloads [37]. All have in common, that they work on integers or that the compressed form is based on integers. In this thesis we will only consider the *compatibility* between error coding and compression schemes, so that we disregard the problem of *selecting* good compression schemes. Furthermore, we will also not consider any special optimizations.

DICT: Variable-length data like strings are “compressed” by encoding the original values using a dictionary which replaces them with fixed-length integer data in the base column. The integer code data can be processed easier and faster [3, 4, 18, 199]. By that, the column contains integers and an additional data structure is required to store the actual strings. The simplest form constructs a dictionary for an entire column or vector, sorted on frequency, and represents values as integer positions in this dictionary [3]. Other possibilities are to use index-structures like trees or to sort the dictionary entries lexicographically [18]. When columns are split into vectors, there may still be a single dictionary per column, or a dedicated dictionary for each vector and dictionaries may also be used across multiple attributes [18]. Figure 4.5 shows an example, with the original table (a), the compressed column which stores only integers (b), and the dictionary entries (c). Here we can also see that the IDs in the dictionary need not be consecutive, which can help to not rebuild the dictionary whenever entries are added, removed, or altered [18]. The storage size of the integers can further be reduced by the other, following techniques. In practice, not just strings are dictionary encoded, but *any* logical data type can be dictionary coded, which makes sense whenever the number of stored *distinct* values is (much) smaller than the whole data type domain. As we noted above, we do not consider string compression and DICT is only a mapping and not a string compression in the strict sense.

RLE: When there are many consecutive data elements with the same value, one can compact their representation by storing only the value and the number of times it consecutively occurs,

VOID	String
0	“Mail”
1	“Air”
2	“Rail”
⋮	⋮

(a) Original String Attribute

VOID	$\text{String}_{\text{Dict}}$
0	30
1	10
2	80
⋮	⋮

(b) Compressed String Attribute

ID_{Dict}	Value
10	“Air”
30	“Mail”
80	“Rail”
⋮	⋮

(c) Attribute Dictionary

Figure 4.5: Example of dictionary encoding (DICT).

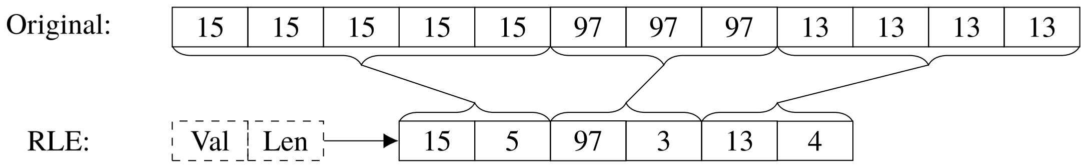


Figure 4.6: Example of run length encoding (RLE).

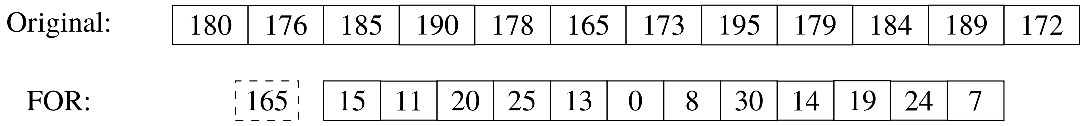


Figure 4.7: Example of frame of reference (FOR) with the smallest value as reference.

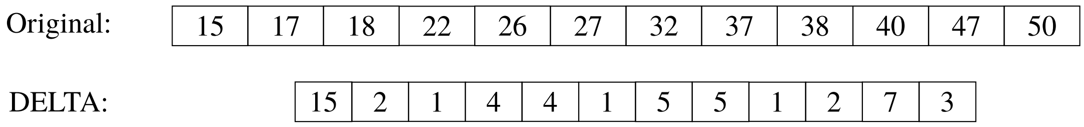


Figure 4.8: Example of delta encoding (DELTA).

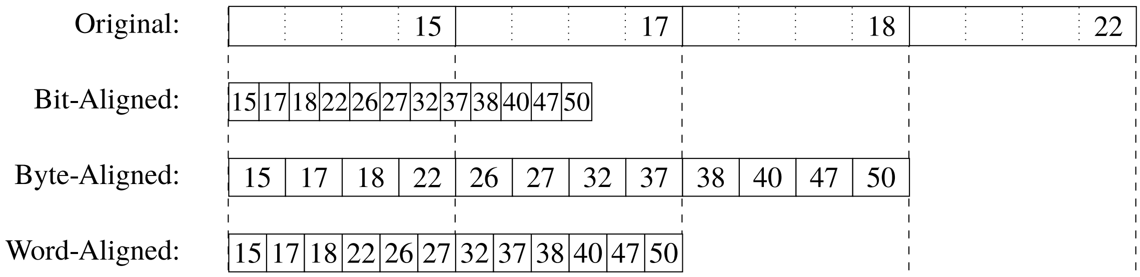


Figure 4.9: Examples of null suppression (NS). The integer sequence is the same as in Figure 4.8.

called a *run* [4, 158] (Figure 4.6). By that, only two (integer) values are stored for each run: (1) the value itself, and (2) the run length.

FOR and DELTA: Instead of storing the values themselves, one can store only the difference of each value to some reference value, which can be e.g. the smallest, largest, or average value. When a single reference value is used for a batch of values, this is called FOR (Figure 4.7). This reference value must be stored additionally to the actual data values. When the minimum value is used, only positive differences are stored. An alternative is to use as reference each previous value, called DELTA (Figure 4.8), which avoids storing the additional reference value. However, this may lead to negative values and DELTA may be better suited for ascendingly sorted data so that only positive (and potentially small) differences are stored.

NS: Null suppression aims at reducing the actually required number of bits used to store values (Figure 4.9). According to Damme *et al.*, there are three classes [194], illustrated in Figure 4.9: (1) bit-aligned, (2) byte-aligned, and (3) word-aligned. The first two classes try to compress integers using a minimal number of bits or bytes, respectively, where both store the values consecutively and without a gap. In contrast, the word-aligned techniques arrange as many integers into word-boundaries as possible, typically 32- or 64-bit words. As can be seen from Figure 4.9, each technique results in a different storage requirement (size) for the integer sequence. Here, the original integers would be stored using 4 bytes, with the byte boundaries indicated by dotted lines. Bit-aligned null suppression uses for a block of integers the smallest number of bits to store all integers in the block. While this results in the smallest memory footprint among the three techniques, we see that integers cross byte- or word-boundaries, e.g. 27 and 37, respectively. In contrast, byte-aligned compression leads to the largest memory footprint, and for word-aligned it is somewhere between the other two techniques. Null suppression techniques can also be categorized into using horizontal or vertical data layouts [37, 108], which is yet another design dimension covered just below in Section 4.1.4.

Vectorized Compression Algorithms: While there already exists a large number of compression algorithms, there is an even larger variety of implementations, while vectorization using SIMD instructions has gained quite some attention [104, 139, 162, 175, 194]. We discussed the importance of SIMD processing in main memory-centric column stores in Section 3.3. For compression algorithms, vectorization improves performance in terms of compression, decompression, and processing speeds [37]. Since this affects Requirement $\mathcal{R}2$, we have to consider later on the compatibility between (vectorized) error coding and compression techniques. For instance, one specific null-byte suppression implementation is the vectorized 4-wise null suppression algorithm by Schlegel *et al.* This algorithm is specialized for compressing four 32-bit uncompressed values in parallel. Each value requires a 2-bit suppression mask to indicate the number of leading null-bytes, which can be at most three in this case, so two mask bits suffice. They employ SSE byte shuffle instructions and generate a one-byte mask, containing the four individual 2-bit masks of the four compressed values.

Encoding may require multiple passes over the original input data for some of the above techniques. For instance, FOR needs to find the reference value for a whole block or column of values in a first pass over the data and then, in a second pass, use that reference value to compute the smaller output values from all input values. Also, DICT *may* require multiple passes, e.g. when the dictionary is sorted. Then, first the sorted dictionary may have to be created and afterwards the string values can

ID	Number of Items	Total Order Prize	Shipping Mode	Order Date
1	5	17 366 547	30	1 508 943 480
2	10	4 692 918	10	1 509 802 620
3	3	19 384 625	80	1 510 310 520
...				
20 bits	5 bits	32 bits	8 bits	31 bits
`uint32_t`	`uint8_t`	`uint32_t`	`uint8_t`	`uint32_t`
		2 dp	CEST	

Table 4.3: A possible physical data representation with implementation types Θ.

Logical	Physical
`TINYINT`	(u)`int8_t`
`SMALLINT`	(u)`int16_t`
`INTEGER`	(u)`int32_t`
`BIGINT`	(u)`int64_t`
`TIMESTAMP`	(u)`int64_t`
`STRING`	(u)`char_t`

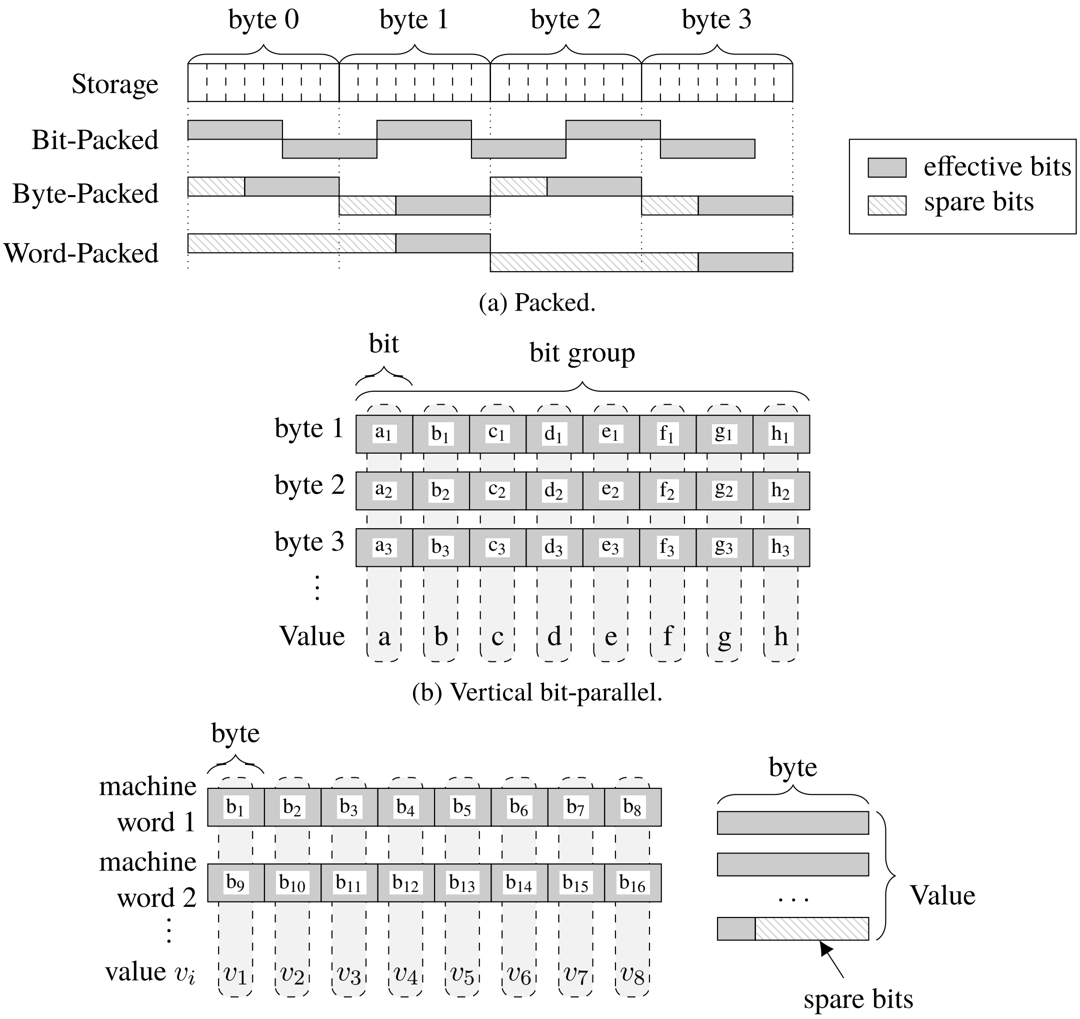
Table 4.4: A possible mapping between logical and physical data types.

be replaced by the dictionary indizes. In contrast, DELTA and RLE can be computed in a single pass, because only consecutive values are considered in both techniques. Finally, for NS it depends on the actual technique. For instance, 4-wise null suppression on its own requires only a single pass. In contrast, SIMD-BP128 [104] requires to first compute the amount of bits needed to store a block of values and then compress and materialize the data in the second pass. When combining logical and physical techniques to a complex compression algorithm, there can be combinations where everything can be done in a single pass (e.g. RLE + 4-wise NS), or where two or more passes are required (e.g. DICT + DELTA + SIMD-BP128). For the latter case, it may also be required to materialize intermediate logically compressed representations in order to apply the next logical or physical technique. These considerations of course also apply to the decoding step. However, this completely depends on the combination of techniques and the actual implementation and exploring this configuration space is outside the scope of this thesis.

To summarize, the goal of lightweight compression is to represent the original data in some small *integer* type. While dictionary coding can be used to store in the columns integer types instead of non-integer types, all of the techniques are used to shrink the integer domain required for storing (most of) the data. Finally, null suppression is used to physically store as few as possible bits or bytes per value. The availability of vectorized lightweight compression algorithms is a perfect fit for in-memory column stores, because the data is naturally aligned in a SIMD-friendlier way than in NSM. By that, our data coding for error control must be compatible with lightweight compression and SIMD, which in turn should allow to meet Requirement $\mathcal{R}2$.

Example Mapping

We now exemplarily map the logical data types from Table 4.1 to physical ones in Table 4.3. To recap, there we have two integer columns, *ID* and *Number of Items*, one decimal column, *Total Order Prize*, one string column, *Shipping Mode*, and finally one timestamp column, *Order Date*, in the respective order. In the row directly below the values, we assign exemplarily an *effective* bit width to the columns, which is the smallest number of bits to represent all values in each respective column. Now, we assume that integer columns are mapped to byte-, word-, etc. boundaries as shown in Table 4.4. This is e.g. the case for MonetDB [21]. W.l.o.g we assume that the decimal column can be represented using 32 bits in this case, so that it fits into the respective physical `int32_t` integer data type. As an additional column parameter, we also need to store the according number of two decimal places, indicated in Table 4.3 below the data type. The strings from the



(a) Packed.

(b) Vertical bit-parallel.

(c) ByteSlice.

Figure 4.10: Physical data layouts.

“Shipping mode ” column are encoded just as in Figure 4.5, where the individual characters are stored in some single-octet `char_t` format like UTF-8. Finally, the timestamp column values can be represented as UNIX timestamps, which fit into 32-bit unsigned integers. As an additional column parameter, the *time zone* could be stored as well, e.g. the Central European Time (CET).

4.1.4 Data Layout

So far, we showed that column stores use a column storage model and that columns mainly contain integers which may be compressed using lightweight compression techniques, with non-integers being converted to integers with the help of dictionaries. Based on that, the issue remains of how to actually layout the individual bits and bytes. First, we could simply store all data items *consecutively* in a bit-, byte-, or word-oriented fashion, as in Figure 4.10a, which reflects a rather natural data layout. On the one hand, byte- and word-packed layouts may introduce additional spare bits. On the other hand, in the bit-packed layout individual values may cross byte- and word-boundaries, which requires a bit more effort to access individual values, since shifting and masking operations are needed. While the byte- and word-packed layouts can be quite obviously accessed using SIMD instructions[9], Willhalm *et al.* showed how to do this for bit-packed data as well [189, 190].

Li and Patel introduced a new data layout which is beneficial especially for sequential scan operations called *Vertical bit-parallel method* [108]. Since their Horizontal bit-parallel method is *in general* like the bit-packed layout we do not consider it here. Also, their BitWeaving methods are specialized variants of the bit-parallel methods and we only concentrate on the basic idea here. In the vertical layout, each of a value's bits are stored separately and the bits of a set of values are stored consecutively. Figure 4.10b shows for an example of eight values the storage, where each i-th byte contains the i-th *bit* of each value. The advantages of that layout are: (1) it inherently treats any number of effective bits, (2) many values are processed in parallel (bit-by-bit), (3) and for filtering data, early pruning is possible which leads to less memory being accessed. Its disadvantages are the costly generation of that layout at load time and the costly re-materialization of individual values. For the former, the individual bits of many values must be extracted and packed together, while for the latter, many bytes must be accessed and bits be shifted to materialize a single value.

Shortly after, Feng *et al.* introduced another data layout which is not only beneficial for scans of whole columns, but also for lookups of individual values [48]. Its idea is a mixture of the previously mentioned horizontal and vertical layouts, shown in Figure 4.10c. There, each value is divided into *bytes* and in the fashion of the vertical layout, the bytes of multiple values are stored next to each other in one *machine word*. This method is very SIMD-friendly, because bytes are the smallest unit for multi-operand instructions and the machine word width can easily be chosen to be 64 bits (x64), 128 bits (SSE), 256 bits (AVX) etc. When the number of effective bits is not a factor of eight, then padding zeros are introduced to fill each value's last byte. In their approach, the effective bits are stored in the MSBs so that the spare bits start at the LSBs.

Regarding the example mapping from Table 4.3 again, we constructed a case where the effective bit width of some columns is smaller than the implementation data type. A concrete DMS could now decide to store the values in a bit-packed or bit-parallel fashion to reduce the storage requirements compared to a naive realization which uses only the available implementation types, i.e. -byte- or word-packed layout. In this concrete example, a tuple requires at least $20 + 5 + 32 + 8 + 31 = 96$ bits when bit-packing all values and $32 + 8 + 32 + 8 + 32 = 112$ bits when byte-packing all values, which is an overhead of $\approx 17\%$. Be reminded that packing (data layout) and aligning (compression) are two different techniques.

4.1.5 Tree Index Structures

Although DSM already yields much better data locality for column scans, index structures are still used as *secondary access paths* for in-memory column stores [39, 83, 100, 102, 148]. The B-Tree [16] and variations like the B*-Tree [84], or the B$^+$-Tree [35] were the most famous index structures for disk-based DMSs [35] and even for in-memory DMSs there are several adaptations [39, 100, 106, 135]. In contrast to the rather simple column layout, tree index structures are typically very pointer-heavy data structures. In the following, we will concentrate on B-Trees, since this is still one of the major index structures. We will first describe the structure of B-Trees and afterwards the basic operations on them. Then, we will discuss changes to the structure and operations which are introduced by major variants.

[9] This holds at least for most of the x86 SIMD ISA variants of SSE and AVX.

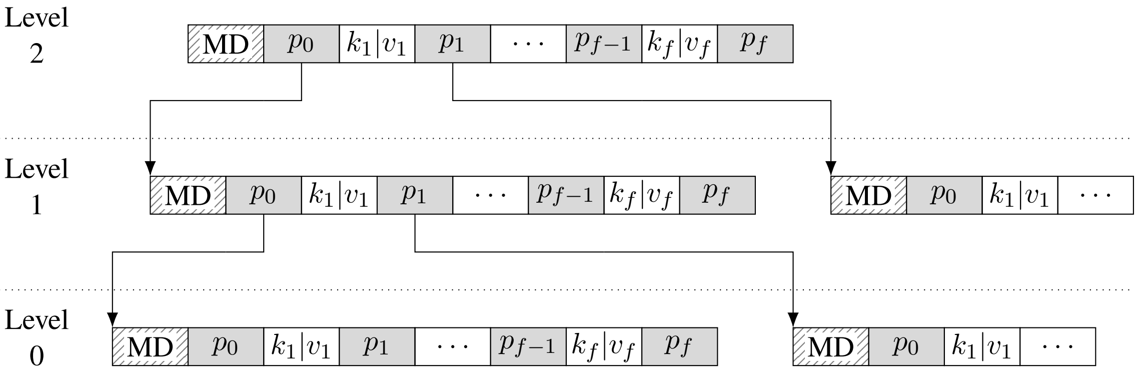


Figure 4.11: Basic B-Tree structure. Pointers are highlighted gray.

B-Trees are tree index structures which store multiple key-value-pairs ($\mathrm{k_x}|\mathrm{v_x}$) per node, as indicated in Figure 4.11. The keys are typically sorted [16] and in the following, we assume an ascending sort order, but all descriptions can be easily matched to other sort orders. Each node typically fills one or a few disk blocks and node sizes of up to 4 KiB are common. A B-Tree of *order* d may contain up to $2d$ keys and in each node, except the root node, there must be stored at least d keys. This means, in each B-Tree node there are between $d \ldots 2d$ key-value-pairs. We denote the actual number of pairs in node i as f_i, the *fill level* of that node, and the set of pairs as $\{(k_0|v_0),(k_1|v_1),\ldots,(k_{f_i-1}|v_{f_i-1})\}$. Next to these pairs, nodes contain f_i+1 pointers $\{p_0,p_1,\ldots,p_{N_i}\}$, i.e. always exactly one more pointer than keys or at most $2d+1$. The j^{th} pointer links to the child node which contains those keys which are in the range between keys k_{j-1} and k_j from the current node. The exceptions are the first and last pointer, obviously, which point to nodes containing keys smaller than k_0 and larger than k_{f_j}, respectively. The values contain the payload, which in the case of unique keys can be inline data, e.g. integers, or an indirection to the actual data like record identifiers (RIDs). When a key may be associated with multiple values, then the value can point to a list of the actual values. For in-memory B-Trees, an additional padding (not shown in the figures) may be used to align nodes to a certain memory boundary, e.g. to align nodes to the native memory page size. Additionally, each node may keep some metadata (MD), typically at the front, for instance to store the node's tree level (cf. levels in Figure 4.11) which is zero at the lowest (leaf) level and largest at the highest (root) level (cf. Figure 4.11), and the node's number of contained keys (fill level). According to Graefe and Stonecipher, many implementations also contain *sibling* pointers among nodes on the same level [64], and compression is also used to reduce nodes' storage footprint [17, 141]. In the following, we describe the basic operations on a B-Tree.

Lookup. The primary goal of index structures is to fasten the lookup of keys and associated values. For a given pair $(k_l|v_l)$, the lookup works top-down from the root and is essentially the same for each node. First, in a node the largest key k_x which is equal to or smaller than k_l is searched. When there is such a key k_x, then we follow pointer p_x to the next child node, otherwise the last pointer p_f is followed. When there is a key which exactly matches the lookup key k_l, the associated value is returned immediately and the traversal ends. When the leaf level is reached but there is still no key match, then the lookup returns some "not found" status.

Batch Lookup. Next to looking up individual keys, index structures are also used to search for *ranges* of keys. Since the keys are stored sorted, such a batch lookup can easily return multiple keys from a node. When the searched range spans several nodes, the parent node(s) are used to continue the search in the siblings.

Insertion. The addition of new key-value-pairs is realized through some *insert* method, which takes as input an appropriate key-value pair. First, the tree is traversed exactly like for the lookup. When the key already exists, the value is overwritten or, in the case of multiple values, appended to the list of values. Otherwise, the new key is inserted into the leaf node where no match was found. When the node is not full, yet, then the key is inserted by moving the larger keys to the right, and then inserting the key-value-pair at the appropriate position. When the leaf node is, however, already full, then that node must be split (*node split*). For that, the list of all keys including the new key is split up, such that the very middle key k_{mid} is pushed to the parent node and the half containing the larger keys are moved into a newly created node. The middle key k_{mid} must be inserted into the parent node, where an additional node split may be required, which works respectively and may propagate up to the root.

Removal. Opposite to insertion, keys can also be removed from the index. As for insertion, the key is looked up, where not finding an exact match results in no action. Otherwise, when multiple keys are allowed, the value is removed from the list. When that list becomes empty, or when only single values are used, then the key-value-pair is removed from the node. When the current node is a leaf, then all the larger keys are moved to the left. Otherwise, in a non-leaf node, the left-most key of the right subtree of the deleted key must be moved to that position. Furthermore, the number of contained keys may become lower than the minimal allowed number d, which requires merging the current node with other nodes (*node merge*). For that, the siblings must be inspected and the nodes' contents must be redistributed such that the restriction on the number of keys is still met for all siblings. This also involves exchanging keys with the parent node. Again, this operation may propagate up to, but not including, the root level, because the root node may contain any number of keys.

B-Tree Variants

It was realized early on that the original B-Tree design was inefficient with respect to all operations. The restriction that there must be at least d keys in a node may in the worst case lead to all nodes being populated only by 50%, which leads to bad storage efficiency, and deeper trees, which in turn results in more traversal steps [16, 35]. Also, for node splits, node merges, or when range lookups span multiple nodes, the parent node is required to access sibling nodes, which may require additional traversal steps.

The B*-Tree variant [84] changes the minimal key requirement such that each node must be filled at least by a proportion of $^2/_3$. The node splitting during insertions is delayed through redistributing keys in the neighbors until 2 nodes can be split into 3, which again are filled by $^2/_3$ each. This increases the minimal storage utilization to 66%, which in turn speeds up lookup operations due to smaller tree heights [35].

In the B$^+$-Tree variant [35], the values are only stored in the leaf level. Furthermore, all nodes in the leaf level contains pointers to their neighbors, resulting in a single- or double-linked list of leaf nodes. By that, inner and leaf nodes have different layouts, because inner nodes need not store values and by that have space for additional keys and pointers, while the leaf nodes need not store any pointers at all. However, batch lookups become cheaper, because the leafs are interlinked.

Some other variants which are tailored to in-memory systems, like the FP-Tree, lift the requirement of sorting the keys in each node, because the sorting overhead may now be a limiting factor [135]. There, nodes become more complex because fingerprints are introduced to reduce the amount of keys against which to compare the search key.

4.1.6 Summary

In this section, we revised the basic storage architecture of main memory-centric column stores. We saw that the ANSI/SPARC architecture provides an abstraction for client applications in the form of logical and physical schemas. The physical schema furthermore describes how data is stored in memory. First, the physical storage model defines that data is managed in the column-oriented DSM data layout, where columns are stored separately. The data representation describes how logical data types are mapped to implementation data types. Here we showed that the columns basically contain only integers, while helper structures like dictionaries store variable-width data like strings. Instead of floating point numbers, fixed point arithmetic is used in the form of decimal numbers. The storage consumption of columns and indexes can be reduced using lightweight compression algorithms, which may be composed of several of the five basic compression techniques. Additionally, there are several physical data layouts which describe how the bits and bytes of the (compressed) values are actually arranged in main memory. Finally, we revised the B-Tree and some of its derivatives as fundamental and wide-spread index structures. In the following, we will discuss how data hardening works together with all these aspects.

4.2 HARDENED DATA STORAGE

Based on the general column store architecture that we revised in the previous section, we can ascertain the following points:

1. The logical data types are only *conceptual* and are not affected by any physical data coding. This is beneficial, because changing the physical schema does not change anything for client applications (cf. Figure 4.1). Due to the abstraction, we only need to change the physical layer from the ANSI/SPARC architecture.
2. The storage model (DSM) consist of two main *data structures*: (1) *data arrays* as primary access path and (2) *index structures* as secondary access path.
3. The physical data representation constitutes of fixed-width integer or decimal data types for columns and index structures. When columns are dictionary coded, additional data heaps store the original values. As we noted earlier, decimal numbers are in the end represented as integers, as well. In the following, we consider *integer* and *string* physical data types.
4. Lightweight integer compression is used to reduce the original data domains and dictionary coding is used to also represent strings as integers in the base columns and store strings in a separate dictionary structure.
5. There are in principal two physical data layouts: horizontal and vertical packed layouts. The former stores the values consecutively next to each other, while the latter rips values apart and stores individual bits or bytes from different values next to each other.

	ID	Number of Items	Total Order Prize	Shipping Mode	Order Date
	985	558	15 299 927 907	6990	1 329 379 205 880
	1970	1170	4 134 460 758	2330	1 330 136 108 220
	2955	351	17 077 854 625	18 640	1 330 583 568 120
A	985	117	881	233	881
$\|A\|$	10	7	10	8	10
$\|\mathbb{C}\|$	30	12	42	16	41

Table 4.5: Hardened integer columns, each with an A tailored to the data representation, extended from Table 4.3. Here, we assume a desired minimal detectable bit flip weight $\eta = 3$.

ID_{Dict}	Value
2330	"Air" `0x5a`$_{16}$
6990	"Mail" `0x29`$_{16}$
18 640	"Rail" `0x36`$_{16}$

Table 4.6: Dictionary with AN hardened IDs and XOR hardened string values.

layout	η 1	2	3	4
bit-packed	$^{106}/_{96} \rightarrow 10\%$	$^{126}/_{96} \rightarrow 31\%$	$^{141}/_{96} \rightarrow 47\%$	$^{160}/_{96} \rightarrow 67\%$
word-packed	$^{184}/_{112} \rightarrow 64\%$	$^{192}/_{112} \rightarrow 71\%$	$^{224}/_{112} \rightarrow 100\%$	$^{240}/_{112} \rightarrow 214\%$

Table 4.7: Overheads for different minimal detectable bit flip weights η and data layouts for our running example from Table 4.5 using the appropriate As from Table 3.6.

6. There are many compromises to be made regarding mainly the storage and processing requirements of the chosen storage model, data representation, and data layout. An error detection technique should integrate seamlessly and put as few as possible restrictions on the available variant space of the physical schema.

In the following, we will discuss how we modify the parts of the physical schema, by means of the error control codes which we selected in Chapter 3. For our hardened storage approach, we consider all of the above aspects. Most of the discussion of this section was published, in particular the discussion regarding hardening integers with AN coding in [87], the combining of lightweight compression and AN coding in [85, 86] and furthermore, we published the considerations for hardened B-Trees in [88] and gave a live demonstration, where we used a heat gun to stress real DRAM memory modules while running queries against our hardened B-Tree [89].

4.2.1 Hardened Physical Data Types

For data arrays with integer values, we harden values using AN coding, which requires only multiplication with the constant factor A (cf. Equation (3.10)). By that, the required amount of storage increases from the data width to the code width. The additional memory consumption must be considered, which may affect the choices for data representation and very likely also the data layout. As we discussed, each DMS may use different mappings from logical data types to the physical data representation. This results in different data widths for each column, or even each vector, when the DMS horizontally partitions columns, e.g. with the PAX storage layout. Since we can set A for each column and vector individually, it can be chosen according to the actual data representation, including lightweight compressed integers. As an example consider Table 4.5, which shows an AN coded physical data representation of the integer columns from Table 4.3. W.l.o.g, for this example we assume a minimal detectable bit flip weight $\eta = 3$, but this in practice depends on the actually assumed error model or other demands and might even vary for the different columns. For our running example, the chosen As (cf. Table 3.6) and the code bit widths are given in the last two rows of table 4.5. The storage increase per tuple, compared to unencoded storage and assuming a bit-packed horizontal data layout, is

$$\frac{30+12+42+16+41}{96} = \frac{141}{96} \rightarrow \approx 47\%,$$

whereas when assuming a word-packed data layout for both uncoded and coded data, the storage overhead is

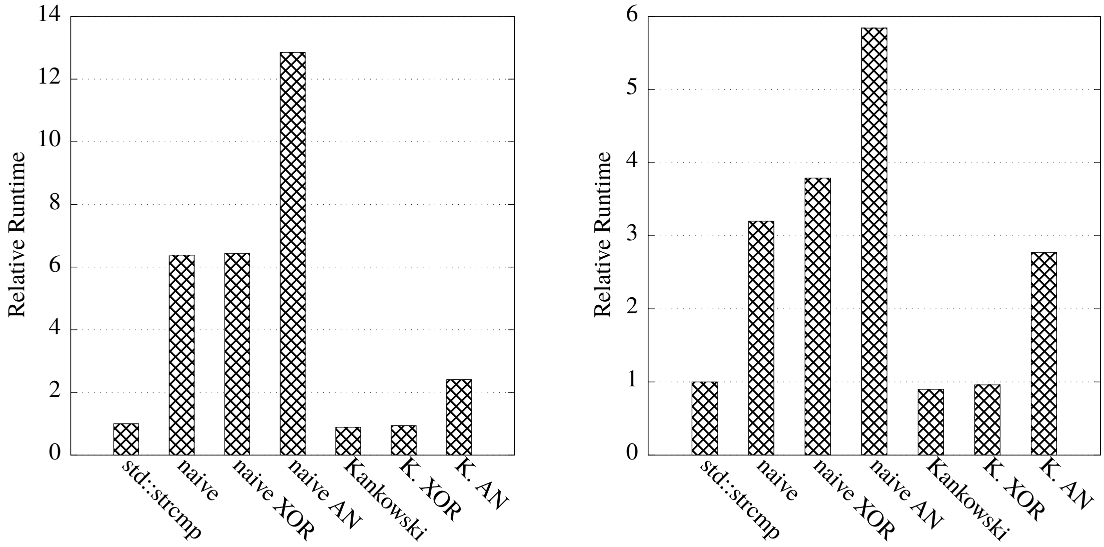
$$\frac{64+16+64+16+64}{112} = \frac{224}{112} \rightarrow 100\%.$$

The overhead varies with varying minimal detectable bit flip weight η and some more examples are given in Table 4.7. While in Figure 3.25 we compared the memory consumption for individual data widths, this includes multiple columns of varying data types but the same η. The numbers are again different when each column is hardened for a different η.

Next, we consider variable-width data types *decimals* and *strings*, for which the case is a bit more complex. Both are based on physical representations which potentially use multiple units for storage: limbs or character encoding units, respectively. We will use the notion of *limb* throughout this discussion referring to both concepts. However, due to their nature, both data types are used quite differently. While strings typically have a fixed encoding using typically single-byte limbs, limb sizes for decimal numbers can vary much more. Arithmetic operations are typically only done on numbers, while pattern matching or filtering are done on both types. Moreover, typically different types of aggregates are computed, e.g. sum, average, minimum, and maximum for decimals, and e.g. shortest, longest, or average number of limbs for strings. Therefore, we will consider both types separately.

For decimals, based on the limb-like representation there are two options for hardening: (1) harden each limb individually, or (2) regard the full value as a huge number and harden it as a whole. Since decimals are numbers, it seems only natural to harden them using AN coding. The first approach requires adapting the algorithms to work on larger limbs, as each limb becomes a code word on its own. Since computer systems have a maximal word size, today typically 64 bits, the additional code bits can further reduce the limb data domain, as we cannot use arbitrarily large limbs. Using the second approach allows to leave the algorithms unchanged, but unfortunately, deriving the detection capabilities for huge arbitrary data widths is very expensive, as we showed in Section 3.2.3. There we noted that, relative to 32-bit data, the effort to compute the probability for a single A for 64-bit data is $4^{64-32} = 4^{32} \approx 1.8 \cdot 10^{19} \times$ as high. For decimals, *currently* only the first approach is feasible, but when determining optimal As for large code widths becomes viable, then the second approach may be a much better alternative, as it primarily leads to much less space overhead. For instance, protecting 32-bit data against $\eta = 4$, we need $A = 32417$ and $|A| = 15$ bits. But when we split it up into four 8-bit limbs, then we need $A = 1939$ with $|A| = 11$ bits, so that in total we have $4 \cdot (8+11) = 76$ bits, in contrast to the 47 bits for the former case, which is an overhead of $76 \div 47 - 1 \approx 62\%$. Additionally, as we have shown in Figure 4.4 and table 4.2, larger limbs generally tend to have less data domain abundance. Finally, coded decimal numbers which have less limbs require less operations when processing them, to care for overflows, underflows, carries, etc. Consequently, this is more a matter of technical feasibility and today we have to use the first, less efficient approach for *large* decimals, while when it becomes viable to compute SDC probabilities for very large data widths, then the second, more efficient approach should be used.

As for decimals, we could consider strings either as a huge number, or more naturally as a sequence of character encoding units, which we will refer to in the following as “limbs”, too. We just



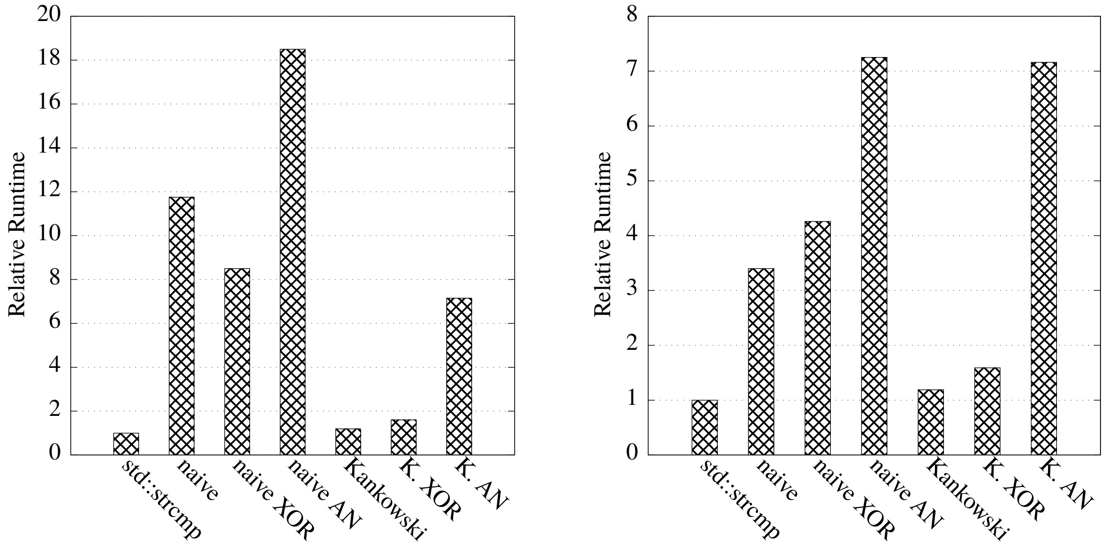
(a) 8-bit limbs, relative runtime.

(b) 16-bit limbs, relative runtime.

Figure 4.12: Relative runtimes for string comparison on system 1.

	`std::strcmp`	Naive	Naive XOR	Naive AN	Kankowski[10]	K. XOR	K. AN
8-bit	1.00	6.36	6.44	12.85	0.89	0.94	2.41
16-bit	1.00	3.20	3.79	5.84	0.90	0.96	2.77

Table 4.8: Numbers for Figure 4.12.



(a) 8-bit limbs, relative runtime.

(b) 16-bit limbs, relative runtime.

Figure 4.13: Relative runtimes for string comparison on system 2.

	`std::strcmp`	Naive	Naive XOR	Naive AN	Kankowski[10]	K. XOR	K. AN
8-bit	1.00	11.75	8.50	18.50	1.19	1.60	6.16
16-bit	1.00	3.40	4.26	7.25	1.19	1.59	4.51

Table 4.9: Numbers for Figure 4.13.

discussed that for huge numbers, AN coding is currently impractical, so encoding a whole string as a single number is infeasible. Also, as we have seen, the optimal code parameter varies a lot for varying data widths so that for the same η, each string might be encoded with a different A, depending on the actual string's length. Encoding each limb (character encoding unit) would be possible, but this would require to adapt all string operations, at least to handle larger limbs. Since string values can be pretty long, checksums make more sense in this case. As a consequence, operations on string values need to compute the checksum in parallel to the string operation. Further, there are some operations which need not work on all characters, like prefix matching. For these, now all characters must be touched to create the checksum, assuming there is one checksum per string value. We measured string comparison runtimes for unprotected, XOR, and AN variants for 8-bit and 16-bit limbs, with the results from measurement system 1 shown in Figure 4.12 and the numbers given in Table 4.8. There, we compare the C++ standard implementation `std::strcmp` against naive (scalar) and SSE4.2 implementations, where the SIMD variants are based on a very efficient implementation by Peter Kankowski[10], denoted by a leading "K.". We measured the string comparison runtime for a total of $64 \cdot 2^{20} \approx 67$ Million limbs, whereas we generate two equal strings so that both are compared until their end. Figures 4.12a and 4.12b show the relative runtimes for 8-bit and 16-bit limbs, respectively. The graph and table values are relative to the standard implementation `std::strcmp`. Figures 4.12a and 4.12b have a quite similar form and we can see that the naive implementations are very slow compared to the vectorized versions, whereas the Kankowski version is even slightly faster than the standard implementation, which is why we used it as a base for the hardened string comparison. As a consequence, the vectorized XOR variant is even faster than the GCC's standard implementation and only slightly slower than the base Kankowski variant. In contrast, the runtimes for the vectorized AN variant are $2.41\times$ and $2.77\times$ as high as the standard implementation for 8-bit and 16-bit limbs, respectively. We did the same measurements on system 2, with the results shown in Figure 4.13 and table 4.9. There, all naive and Kankowski variants have higher relative runtimes than for system 1. For system 2, The Kankowski algorithm is 19% slower than the GCC implementation and also the K. XOR variant is much slower relative to `std::strcmp` with 60% longer runtimes. The K. AN variant's performance drops to $6.16\times$ and $4.51\times$ relative runtime. In essence, from a throughput perspective this shows that for both systems, XOR is much better suited for protecting the actual string values. Table 4.6 displays the hardened example dictionary for our example mapping in Table 4.5. Here, the dictionary IDs are hardened using AN coding such that they match the IDs stored in the Shipping Mode column. The string values are appended by an XOR checksum, which is shown as hexadecimal number below each string.

Consequently, to harden the columnar base data, we use AN coding for integer data. Since strings are typically dictionary encoded, the column actually contains encoded IDs or offsets, which in turn point into the additional dictionary data structure, which contains the actual XOR encoded string values. This is a performance tradeoff, to better satisfy Requirement $\mathcal{R}2$. In contrast, since XOR checksums have a fixed structure and offer little adaptability, AN coding might be the better choice when Requirement $\mathcal{R}1$, (effectiveness) and Requirement $\mathcal{R}3$ (adaptability) are more important.

4.2.2 Hardened Lightweight Compression

We will now consider the interplay between the presented lightweight compression techniques and error coding techniques. At first sight, these are contradicting goals, because compression reduces the data size, while error coding increases it. However, both can be elegantly combined so that the impact of hardening is smaller, compared to the raw base data size. Furthermore, as we can see

[10] See `https://www.strchr.com/strcmp_and_strlen_using_sse_4.2`.

from Tables 3.6 and A.1 to A.4, the smaller the data size, the smaller the A to achieve the same error detection guarantees for η. Consequently, compression can have a *double* positive effect. We will now consider how hardening and compression are actually combined.

As we described above, the *logical* techniques DICT, RLE, FOR, and DELTA are basically mappings between one source data domain and a typically smaller target integer domain. Consequently, it seems quite natural to fit with AN coding and Figure 4.14 illustrates how to combine these two. We assume that for compression we use both logical and physical compression. In Section 4.1.3, we discussed that multiple passes over the input data may be necessary for a complete compression algorithm. For the following, we assume that such sub-steps are comprised in a single logical compression step. On a high level, combining lightweight compression and hardening is a two-pass approach, meaning we must iterate at least twice over the uncompressed data.

In the first pass, we have to determine the bit width of the data which would result after applying the chosen logical compression technique(s). As we discussed in Section 4.1.3, such preprocessing may be necessary anyways. Based on that bit width, we choose the appropriate A for the data after all logical techniques, e.g. from Tables A.1 to A.4. When a logical compression scheme divides the original input data into multiple blocks of potentially different compression ratios (e.g. SIMD-BP128 [104]), then for each of these blocks a different A *might* be chosen. This then again depends on the logically compressed data bit width per block. In total, the first pass serves to determine the required parameters for compression and hardening.

In the second pass, the actual compression takes place using the parameters determined in the first pass. While there are four individual steps shown in Figure 4.14, implementation-wise these might of course be combined into a single one, depending on the chosen compression techniques. The considerations for multiple passes from Section 4.1.3 apply respectively. This means, it *might* be required to materialize intermediate data after logical compression or hardening. For RLE-, FOR-, and DELTA-like techniques this results in materializing hardened value-length-pairs, hardened reference values and hardened derived values, respectively. It may also be required to harden metadata for the logical compression schemes, for which different As than for the compressed data might be chosen, again e.g. from Tables 3.6 and A.1 to A.4. For RLE, the metadata comprises the run length of each value, while for FOR-like techniques the reference value is the metadata. In contrast, DELTA does not use any metadata. For DICT-like compression, the IDs or offsets are hardened, which must also be respected by the dictionary implementation. Afterwards, the hardened, logically compressed data is materialized using a *physical* compression technique and data layout, where we are again free to choose from those presented before in Section 4.1.3. This takes place in the form of null suppression, where the techniques must consider the now *larger hardened data size*. Finally, since the null suppression techniques typically need some metadata

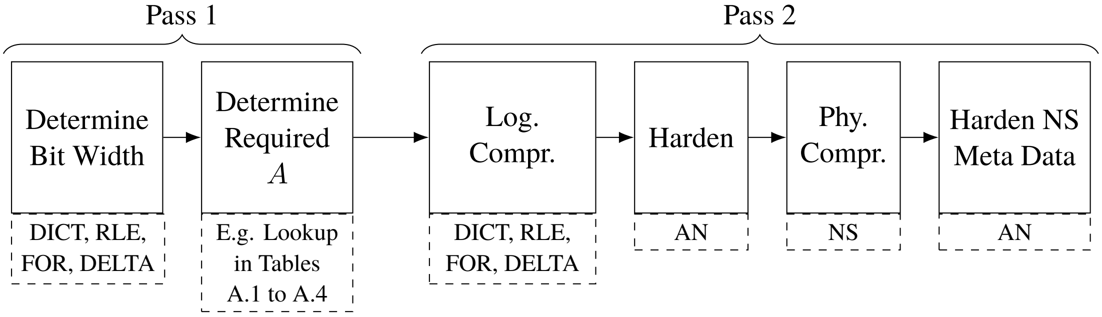


Figure 4.14: The combined process of hardening and lightweight compression. Decompression works in reverse order.

```
1  compress_AN_NS (in elements[], out buffer[], in $A_{\text{Values}}$, in $A_{\text{Mask}}$) {
2    for (i = 0; i < |elements|; i = i + 4) {
3      n1 = elements[i] · $A_{\text{Values}}$;
4      z1 = count_leading_zero_bytes(n1);
5      ...
6      n4 = elements[i+3] · $A_{\text{Values}}$;
7      z4 = count_leading_zero_bytes(n4);
8      mask = (z4 << 6) | (z3 << 4) | (z2 << 2) | z1;
9      buffer ← mask · $A_{\text{Mask}}$;
10     buffer ← n1;
11     ...
12     buffer ← n4;
13   }
14 }
```

Listing 4.1: Pseudo code for AN encoded NS compression. `elements` is the (logically compressed) input array while `buffer` is the output array. `|elements|` denotes the array's number of elements.

again for later decoding, that metadata must also be hardened in the last step. Since this is comprised of integers again, we assume to use AN coding also.

In order to decompress and decode hardened, lightweight compressed data, the previously described process is reversed. It starts with decoding the null suppression metadata, then follows the physical decompression, then decoding of the hardened, logically compressed data and metadata, and finally the logical decompression.

To give a concrete example for 4-wise byte null suppression by Schlegel *et al.* [162], consider the pseudo codes given in Listings 4.1 and 4.2 for hardened compression and decompression, respectively. We can assume, w.l.o.g, that the input data may be logically compressed already. Furthermore, we assume that this is only the second pass, i.e. that the required parameter A was already determined for both values (A_{Values}) and masks (A_{Mask}). Encoded compression for NS works the following. There are input and output arrays to function `compress_AN_NS`, where `elements` stores original data and `buffer` receives the compressed and encoded data. Four data items are processed in each loop iteration (line 3). First, each item is multiplied by A (lines 5-7) and afterwards the leading zero bytes are counted (lines 8-10). This can be done by counting the leading zero bits using compiler intrinsics (`__builtin_clz()` for GCC) and then dividing by 8. The bit compression mask contains the number of leading zeros. It is computed by ORing the lower 2 bits of the zero byte counts together (line 11). Finally, the mask and the compressed encoded words are stored in the output buffer (lines 12-15). Assuming a little endian system, the leading zero bytes of a compressed value are inherently overwritten by the next appended data item, by advancing the write pointer by the number of non-zero bytes of the item just written. Schlegel *et al.* note in their evaluation that loop-unrolling is very beneficial to avoid data dependencies. In their setup, an unroll factor of four was the best one. The algorithm in Listing 4.1 can be adapted easily to four-wise loop unrolling as follows: First, compute elements `n5`, … , `n16` after `n4`. Then, compute `z5`, … , `z16` after `z4`. Afterwards, compute and store the three additional masks after `mask` and, finally, write elements `n5`, … , `n16` after `n4`.

Decompression is also straight forward, as shown in Listing 4.2. A loop iterates over the input buffer which was generated by the `compress_AN_NS` function (line 3). First, the compression mask is loaded (line 5). Then, the number of non-zero bytes – denoted by the mask's lowest 2 bits – of the first data item is stored (line 7) and the according bytes are stored into `item` (lines 8 and 9). The restored item is checked using the inverse A^{-1} and errors may be handled appropriately, e.g. we here call a function `error()` (line 9). We leave the concrete implementation of such a function to the actual DMS. Then, the decoded data item is stored in the output array (line 10) and the read

```
 1 decompress_AN_NS (in buffer[], out elements[], in $A^{-1}_{\text{Values}}$, in $d_{\min}$, in $d_{\max}$, in $A^{-1}_{\text{Mask}}$)
 2 {
 3   for (i = 0; i < |buffer|; ) {
 4     mask = buffer[i] · $A^{-1}_{\text{Mask}}$;
 5     if (mask > 255) error();
 6     i = i + 1;
 7     for (k = 0; k < 4; ++k) {
 8       leading_zero_bytes = mask & 0x3;
 9       item = buffer[i] & (0xFFFFFFFF >> (non_zero_bytes * 8));
10       dec = item · $A^{-1}_{\text{Values}}$;
11       if (dec < $d_{\min}$ || dec > $d_{\max}$) error();
12       elements ← dec;
13       i += 4 - leading_zero_bytes;
14       mask >>= 2;
15     }
16   }
17 }
```

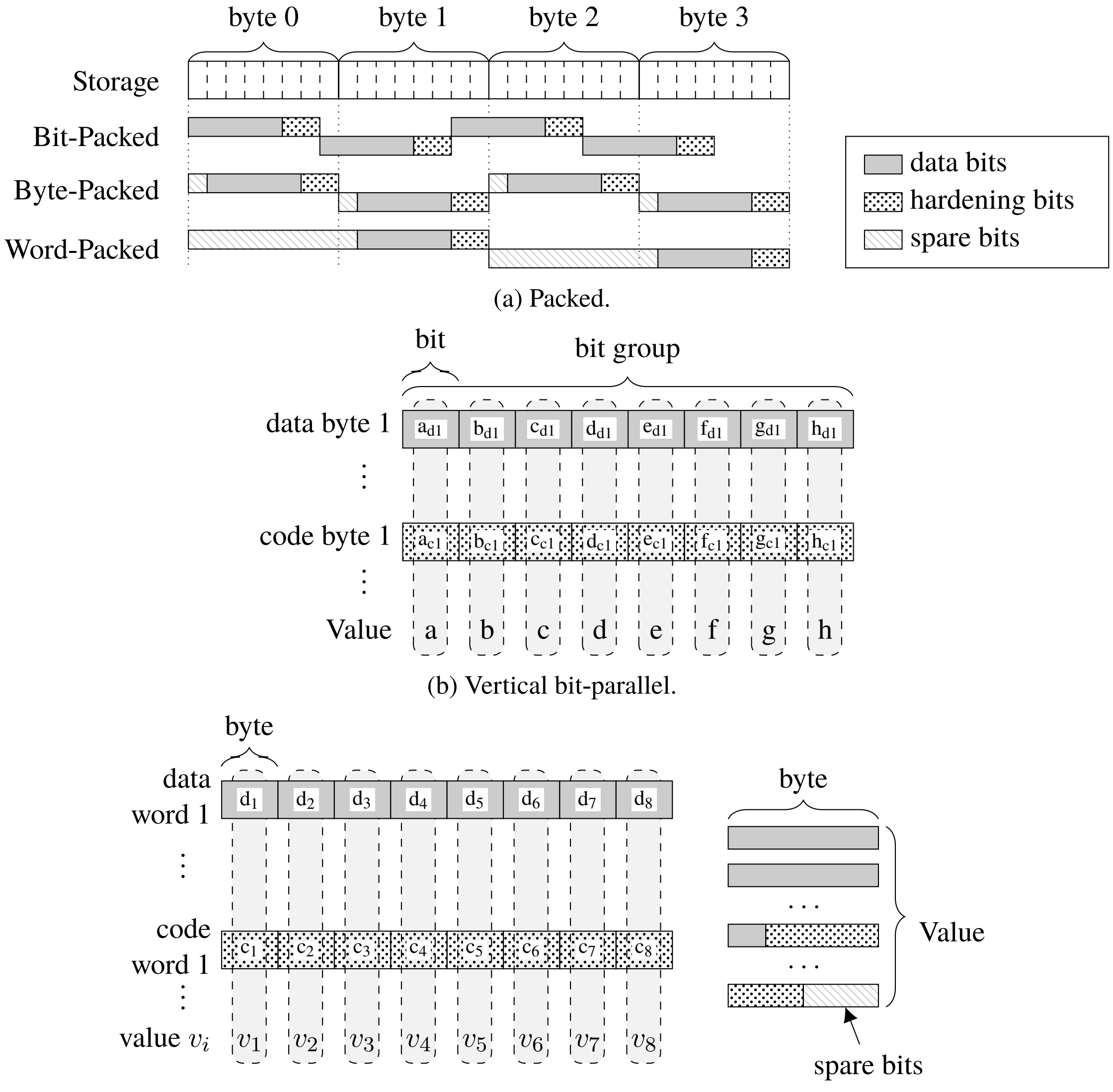
Listing 4.2: Pseudo code for AN encoded NS decompression on unsigned data. `buffer` is the hardened compressed input array while `elements` is the output array containing the uncompressed (or logically compressed), decoded items. `|buffer|` denotes the array's number of bytes.

position of the input buffer is advanced by the number of non-zero bytes (line 11). Finally, the mask is shifted right (line 13), so that the same steps (lines 7 to 13) can be repeated for the next 3 items, since always 4 items are represented by a single-byte compression mask. The four-wise loop unrolling optimization can be applied as for function `compress_AN_NS`.

As we showed above, AN coding can be well combined with integer lightweight compression. The crucial point is that AN coding must be applied in between the logical and physical compression steps. Furthermore, AN coding can also be used to harden all the metadata needed for compression and decompression. Our example Listings 4.1 and 4.2 show that this integration is a straight forward process and that a tight coupling is possible to avoid intermittent materialization.

4.2.3 Hardened Data Layout

Having compressed the hardened data, we need to choose a physical data layout. Here, we basically have the same choices as in Figure 4.10 and the three hardened variants are shown in Figure 4.15. It is very hard to use BitWeaving (Figure 4.15b) or ByteSlice (Figure 4.15c) for hardened data, because the problem here is that to decide whether a code word is valid or not, the whole code word must be read. This is no problem for the horizontal layout (Figure 4.15a), because the code words are stored as one unit. In contrast, the vertical layouts tear apart all the individual bits or bytes. Since we encode each value individually, a vertical layout would require to re-construct each value for error detection. One possibility to mitigate this overhead would be to encode not each value, but each byte or bit group (c.f. Figure 4.10b or in [108, Figure 9]). Then, each bit group can be quickly decoded and the original processing take place. This leads to reduced scan performance, because the effective bit groups are smaller due to the hardening. We could also encode the bit groups such that the encoded bit groups store half as many bits. Then, we can decode and error check two encoded bit groups and combine them to a single bit group of the original size. Then, however, the memory footprint is increased by a factor of two. These are just the two extremes and by shifting the decoded bit groups around, we can also use intermediate configurations where parts of multiple decoded bit groups are combined to result in a bit group of original size. For the hybrid ByteSlice layout, the same considerations apply. There, we could encode a set of bytes such that it fits into a machine word, again. We leave more detailed investigation for future work, while for the following, we assume only a horizontal physical data layout.



(a) Packed.

(b) Vertical bit-parallel.

(c) ByteSlice.

Figure 4.15: Physical data layouts for hardened data.

4.2.4 UDI Operations

Up to now, we discussed how data is *placed* inside the internal layer of a DMS (cf. Figure 4.1). Now, users or other DMS components want to be able to actually *work* with that data, which needs to be added to, updated in, and retrieved or deleted from a database. In the internal layer, we are mainly interested in Insert, Delete, and Update – the UDI operations. Modification can be modeled as a sequence of delete and insert, known as main-delta- or insert-only-architecture [140], which in turn is the standard for main memory-centric column stores. We here discuss in how far hardening affects these UDI operations. For that, we need not know how a user or database component actually formulates a request for performing a UDI operation. On a high level, we can assume that they modify databases in terms of modifying *tuples*. Through the abstractions in a DMS (e.g. external and conceptual layer), the details are hidden from users and partly also from other internal components of a DMS. In the end, this ultimately boils down to modifying bits and bytes, depending on the actual physical data representations which are managed by the internal layer. This happens based on all the aspects described above: data structure (e.g. column or index), physical data types, compression, data layout, and hardening technique. As should have become clear in the above discussions, the hardening techniques seamlessly integrate with the UDI operations. This means, they are *orthogonal* to UDI operations and do *not* affect them. Especially regarding the main-delta-architecture, any modifications are done in a separate delta structure and eventually merged into the main part, where modified parts are rebuilt anyways.

4.2.5 Summary and Conclusions

In this section, we presented our hardened storage layer. As we discussed, we only need to adapt the lowest internal layer, so that the conceptual and external layers can still provide the abstraction towards client applications, without being changed. Regarding the physical data types, since we only deal with integers in the base columns, we harden all integers using AN coding. We then discussed the problems of variable-width, limb-based representations of, e.g., decimal and string values. The limbs can be seen as individual integers again, so that we have several choices here: encode such a variable-width value as a whole, or harden each individual limb, both using AN coding, or use XOR checksums over the whole value. We argued that the first choice, hardening such large numbers with AN coding, is currently infeasible, since on the one hand we did not yet obtain golden As for data widths over 32 bits. On the other hand, each such value might be of different size, which could potentially require to use a different A for each value and this information needs to be stored somewhere, again, and can be corrupted. Hardening each limb individually requires to adapt the algorithms for working with the data. For decimal numbers, we argued that hardening each limb individually, instead of the number as a whole, using AN coding needs to be used as a technical limitation today. When golden As for larger data widths are computed, the first approach can be tested for feasibility, which we leave to future work. For string values, we showed that limb-encoding is also possible and compared this against the third method of using XOR checksums. This latter method stores one checksum per string value, which is not necessarily the same size as the limb size. We could show that the XOR method is faster for the string comparison operation than AN coding, and argued in favor of Requirement $\mathcal{R}2$ to use the XOR method for strings. When effectiveness is of greater concern, then AN coding might be the better choice, due to it providing minimal detectable bit flip weight guarantees. Regarding the physical layout of hardened data, we discussed that the vertical and hybrid layouts are problematic due to high tuple reconstruction costs, since they distribute the bits or bytes of a value across many machine words. Consequently, we only use the horizontal data layout.

4.3 HARDENED TREE INDEX STRUCTURES

In contrast to simple arrays, tree index structures are special in the sense that in addition to data they also exhibit *structure*, which only the application knows about. This means, that, in addition to the data, we also have to detect bit flips which change or destroy this specific structure. First, we will present related work which deals with detection of errors in index structures and in B-Tree variants in particular. Afterwards, we will discuss why these techniques do not suffice for our thesis error model (cf. Definition 4). Finally, we will then describe our online bit flip detection approach for B-Trees. This work was published in [88] and a live demonstration was given on the BTW 2015 conference [89].

4.3.1 B-Tree Verification Techniques

In 1984, Küspert described methods to detect errors in the structure of B-Trees during normal operation, where he introduced pointer sanity checks and key validation techniques [92]. Before interpreting the actual page contents, page header information is used to determine whether this page belongs to the same B-Tree through a table identifier, and whether it is actually a level below the parent. Through the use of forward and backwards pointers, the links between parent and child nodes and also between sibling nodes can be checked. For instance, when traversing upwards, the parent node must actually contain the link to the child node. When traversing downwards, the child node must point back to the originating node as its parent. The sibling links in a child node are verified against the neighbor links in the parent node. The left- and right-most nodes of each inner level must not have a link to a left or right neighbor, respectively. Additionally, he proposed key validation in the inner nodes, but not in leaf nodes. Keys in inner nodes are redundant, because they are derived from the lower layers and only work as key range separators. This fact can be used to compare the key ranges against the parent node's separator keys or against sibling nodes' key ranges. In the leaf nodes, he describes, at most validation of the sort order is possible. From that it is easy to see that in trees where only unique keys are allowed, this uniqueness property can also be verified. In his work, Küspert assumes disk-based database systems, where partial writes occur or where nodes' pages are not updated at all. However, these techniques were deemed lightweight enough to be carried out in each tree traversal as local tree verification.

In the 1990s, Mohan discussed how B-Tree pages, and database pages in general, can be corrupted due to partial writes which stem from performance optimizations in the SCSI standard [119]. He developed error prevention techniques through special page modification and verification after any page read, logging, log analysis and other operations.

Graefe and Stonecipher introduced B-Tree verification mechanisms which are more heavyweight than those in [92]. They argue that *complete* verification of the whole tree is required and they assume that a tree that is to be verified is exclusively locked so that there are no concurrent transactions using that index. By that, tree verification is done in a separate maintenance operation, i.e. offline and not online. Another assumption is that there are typically multiple B-Tree indexes per relational table and that these multiple indexes can be verified first on their own and then against each other. Their first technique, called "in-page verification", uses one or several checksums to detect partial reads or writes. Since usually the database page size (8 KiB in [64]) differs from the disk sector size (e.g. 512 Bytes), a checksum over only a single value per sector is assumed to be sufficient. Next, they propose different techniques for complete verification of the remaining consistency properties of an index. This covers the verification of links between parent and child nodes, as well as between siblings, and key orderings and validity of key ranges. The first technique is the naive index navigation, similar to those from [92]. There, the index is navigated in breadth-first or depth-first fashion. The second approach is called "aggregation of facts" and is particularly useful to support terabyte databases. There, the same facts as before are collected, but they are not immediately verified. The basic idea is that for one found fact, there must always be exactly one "counter fact". For instance, in a parent node, the facts "this (parent) node points to that (child) node" are gathered. Later, when the child nodes are visited, the according counter facts are assembled, like "this (child) node points back to that (parent) node". At the end of the verification step, there must be no fact left, otherwise there are inconsistencies in the index. Graefe and Stonecipher argue that the most important benefit of this approach is, that each node or page is read only once. One problem there is the validation of key ranges between cousin nodes, which are neighbor nodes on the same level with different parent nodes. Fence keys can be used as a solution, which in practice are the parent node's left and right separator keys and which can replace the sibling pointers. Graefe and Stonecipher argue that an additional advantage of their techniques is, that they share lots of code with other tree operations, which is beneficial with regard to source code maintenance, but this does not play a role here. The next verification technique employs bit vector filtering and is actually an optimization of the gathering of facts. There, for each node and child pointer, the combination of index identifier, page identifier, index level, and low and high fence keys is hashed to produce the position in a bit vector. Upon encountering a fact, that bit position is computed and that bit is inverted. After the verification, the bit map should be in its original all-zero state, otherwise errors were detected. In contrast to the other techniques, this one can not pinpoint the error location and a second pass is required upon detection of a corruption. Their final technique is to use the standard query processing facilities and to formulate queries to find corruptions. The advantage is that complex expression can be evaluated, that views can be compared against base tables, when columns are derived using arithmetic and functions, or when dealing with user-defined types.

Graefe *et al.* further extend B-Trees to include information in each node which allows to repair itself. In essence, each such self-repairing B-Tree stores backup pointers to the previous version of its respective page or node, as well as log sequence numbers which point into the recovery log. This idea is taken from previous work on the newly introduced single-page failure class [62], where a database keeps a global *page recovery index* with the same auxiliary information which is now put into B-Trees themselves.

4.3.2 Justification For Further Techniques

In contrast to the above approaches, we do not assume partial reads from or writes to main memory as the error model, but bit flips which occur as an arbitrary effect. Furthermore, offline verification as in [64] contradicts with our Requirement $\mathcal{R}4$ by which we require bit flip detection to occur in an online fashion. Another big difference is the assumption of whether the index is memory resident or not – some techniques are discarded in the related work, because they require additional disk I/O, which is very costly. Also, while B-Trees in disk-based systems are optimized towards the I/O characteristics of disks, like a certain page size of several disk sectors, in-memory indexes are tailored to the characteristics of the memory and cache subsystems. Other optimizations for detecting tree corruptions, like the bit vector filtering [64] or auxiliary information for self-healing databases [62] and indexes [63], assume that main memory is reliable and that, consequently, the auxiliary information for error correction is also reliable. However, in the view of transient bit flips in main memory, this assumption simply does not hold any longer. In particular, for self-healing data structures the pointers to backup pages and the log sequence number can become corrupt as well and such errors must be detected, too. Therefore, we introduce a different *online* hardening approach which is embedded in the tree traversal and other operations on such a tree index. In the following, we concentrate on the still widely used B-Tree and develop a variant which we call *Error Detecting B-Tree (EDB-Tree)*.

4.3.3 The Error Detecting B-Tree

We will now describe the node layout of our baseline B-Tree, which we extend with error detection capabilities afterwards. Its physical layout is shown in Figure 4.16, which shows 3 inter-linked nodes whereas black and red arrows denote links from a node to its children or from a node to its parent, respectively. Each node is structured the following. At first, there is a pointer P to the node's parent node which is *null* in the case of the root node. Then, there follows a 2-Byte unsigned

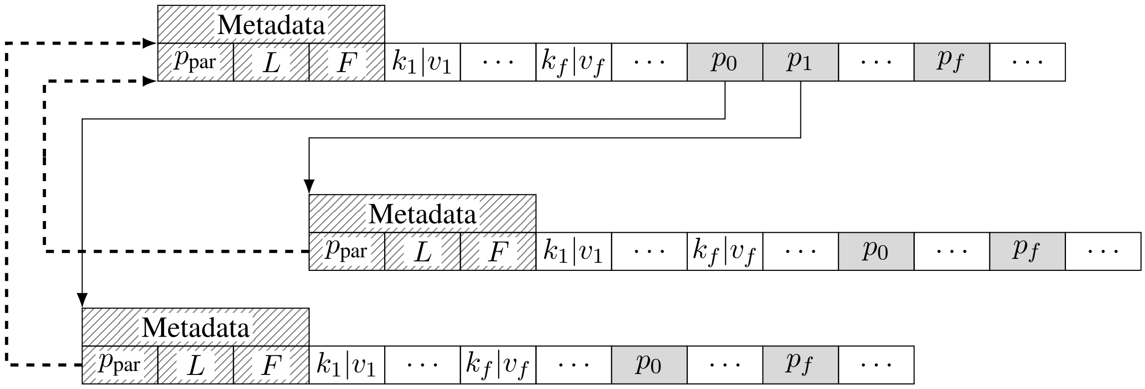


Figure 4.16: EDB-Tree node layout.

integer L denoting the node's *level* inside the tree, which is 0 at the leaf level and increases by one at each higher level towards the root. Afterwards, another 2-Byte signed integer F (fill level) represents the number of key-value pairs currently contained in the node. On the one hand, using signed integers "sacrifices" one bit, but on the other hand, it allows to check for some memory error as is explained below. At next, there are the key-value pairs $\{(K_1, V_1), (K_2, V_2), \dots\}$ where each key is 4 or 8 bytes wide as well as each value, which e.g. can be inline data, a RID (record identifier), or a pointer to the actual data. Inside a node, keys are kept in ascending order. Finally, there follow pointers to child nodes $\{P_1, P_2, \dots\}$, which are always 8 bytes long. Finally, a padding – not shown in the figure for conciseness – allows to align the node to a desired boundary – usually the system's memory page size of 4 KiB. Note also that the compiler (GCC) aligns L and F so that they together consume a total of 8 Bytes for better alignment of the subsequent node members. As is common for B-Trees (and its derivatives) parameter K delimits the number of key-value pairs and pointers in nodes: except for the root, in a node there must be present $K \dots 2K$ keys and accordingly up to $2K + 1$ child pointers. For 4 byte keys and values K is 127 (254 keys and 254 values per node), while for 8 byte keys and values K is 84 (168 each per node). Accordingly, there are up to 255 or 169 children for a node, respectively. For mixed data widths of 8 byte keys and 4 byte values or vice versa K is 101. Table 4.10 summarizes the values for K according to the data width of keys and values for our B-Trees. For the baseline B-Tree, only each node's fill level is checked to be at least zero and at most $2K$, as this is the only variable messing up node scans. Otherwise, this baseline employs no error detection at all. Corrupt pointers to unallocated memory result in segmentation faults, while corrupted keys and pointers to the wrong child may result in false positives or false negatives – keys which were never added to the tree are found and the other way around, respectively. All variants presented hereafter inherit the fill level check and for all tree variants methods for building the trees and doing point queries are fully implemented. W.l.o.g, we store values inside all inner and leaf nodes, instead of only in the leafs. All our error detection techniques also apply to those B-Tree variants which have other properties like storing values only in the leaf nodes.

Key width [Bytes]	4	8	4	8
Value width [Bytes]	4	4	8	8
K (B-Tree, EDB-Tree, EDB-TreePB)	127	101		84
K (EDB-TreeCS, EDB-TreePBCS)	126	101		84

Table 4.10: K for different key and value sizes for baseline B-Tree and the EDB-Tree variants.

In order to cope with an increase in transient error rates in future hardware, our idea is to extend the existing online verification methods towards detecting bit flips during tree traversal and node scans. Assuming that the whole index is memory resident, other checks can be included. Since we propose to make B-Trees detect main memory errors, we call these variants *Error detecting B-Trees (EDB-Trees)*. Furthermore, our premise for the following is that main memory may become corrupt, whereas CPUs – cores and on-chip caches – are reliable components not introducing further errors to data or during computations (cf. Definition 4).

As a first step, we introduce additional pointer sanity checks. For instance, additional pointer checks include checking the alignment of a pointer, i.e. whether it directs to the start of a tree node, which can be done even before following the pointer. Furthermore, assuming an in-memory DMS, we exploit virtual memory management:

1. the virtual address space is much larger than typically employed amounts of main memory,
2. hardware supports less than 64 bits for addressing, since today usually 46–48 memory address lanes are fused on motherboards, and
3. operating systems also have practical limits on the available addressable amount of main memory. For Linux, up to kernel 4.14, 48 bits (256 TiB) were used for virtual addresses and 46 bits (64 TiB) for physical ones[11]. Since kernel 4.14, 5-level paging is supported, which uses 56 bits (128 PiB) for virtual addresses and 52 bits (4 PiB) for physical ones.

For utilizing the much larger virtual address space, EDB-Trees allocate nodes at successive virtual addresses. Consequently, from the process's perspective, the tree consists of a single contiguous memory area. This can be done, e.g. in Linux-based systems, through the use of the `mmap` function, where the DMS can force memory allocation in a specified part of its virtual address space. Nowadays, in pointers the upper bits which would be zero anyways are used for further status information, but with regard to arbitrary bit flips in main memory this information again must be guarded against corruption and specially handled. Consequently, we decide against using those bits in pointers for other purposes. This memory allocation allows to detect corrupt pointers pointing out of this area into unallocated virtual memory space, since we know starting address and length of the contiguous memory area. This is closely related to the table identifier verification proposed in [92], but exploits the virtual memory mechanism.

During tree traversal, three different pointer sanity checks are employed with minimal computational overhead: alignment, memory region and parent-child-relation. Since the node size is fixed, pointers must be aligned accordingly, i.e. for 4 KiB nodes, the first 12 bits must be zero. Knowing the memory area's offset and size, the pointer is then range-checked, i.e. it must point into the allocated region. For handling pointers to the wrong child, the child's parent pointer is compared against the node's address from which we descended. If any of those three checks fails, we can be sure some bit flip(s) occurred. While the alignment- and region-checks suggest an error in the checked pointer, the parent-child relation only indicates one of the two has changed. These sanity checks imply no additional memory footprint. However, they can detect only some errors in pointers where the position of the flipped bit – starting from 1 at the least significant bit – is greater or equal than $ld(\text{node size})$ and smaller or equal to $ld(\text{allocated memory})$. For instance, such an EDB-Tree using 4 KiB nodes and having allocated 4 GiB of memory can not detect bit flips in pointers between bit positions 13 and 32, inclusive. Consequently, about 69% of bit flip patterns could be detected in 64-bit pointers by those sanity checks.

[11] See `https://kernelnewbies.org/Linux_4.14#Bigger_memory_limits` and `https://lwn.net/Articles/717293/`, visited 27 March 2018.

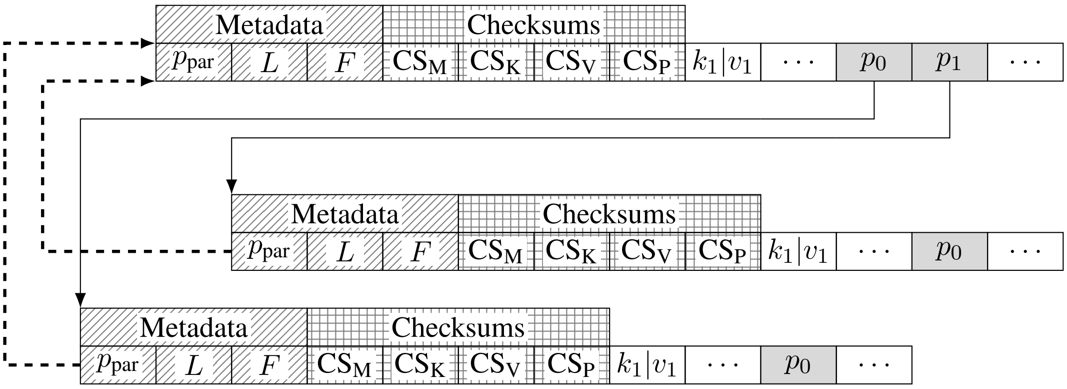


Figure 4.17: EDB-TreeCS node layout.

The following variants all inherit the pointer sanity checks. Since main memory bit flips may occur at any position, any information added to nodes as some kind of redundancy for error correction must be guarded against main memory bit flips, too. Therefore, the following techniques only add little to no redundancy – as e.g. in contrast to [92].

EDB-Tree With Parity Bits

For a given data word, a parity bit guarantees that there is always either an even or odd number of ones. By that, any odd number of bit flips can be detected, while even numbers of bits flips cancel each other out. We call this EDB-Tree variant *EDB-TreePB*. Parities can be either even or odd and in principal there is no difference in employing either. Parity bits are single-bit XOR checksums, as discussed in Section 3.1.2 and the appropriate SDC considerations from Section 3.2.2 apply. Compared to e.g. 64 bits, a single bit causes a memory footprint of 1.6%, while for 32 bits it is 3.1%. Computing a parity bit on modern processors can be done with native instructions in a few cycles[12]. Consequently, parity bits provide limited bit flip detection with very little memory and performance impact. In this EDB-Tree variant, for each node member (cf. Figure 4.16), a parity bit is computed and set as its most significant bit (MSB). Thus, for computing the parity bit, only all but the MSB are regarded and the according member's domain is reduced by one bit. For pointers, this is no limitation since the virtual address space is much larger anyways, as we argued earlier. The parity bit is set for a key and a value, whenever a new key/value-pair is added, or when a value is updated. It is set for a node's fill level whenever that changes, and on pointers when these change in the course of split, merge and delete operations. For each tree traversal, node members are first validated against the parity bit before their first use.

EDB-Tree With XOR Word Checksums

For detecting more bit flips we employ word XOR checksums to groups of node members. An advantage of using XOR checksums is that they allow to correct a single data element, when there is only one corrupted element and when it is known exactly which element is the corrupt one. To improve this rate for a single node keys, values and pointers can be partitioned and one checksum for each partition can be added to the node, which however further increases the memory overhead

[12] Counting bits in a 64-bit word typically has a latency of 3 cycles and a throughput of 1 cycle, meaning that a single count takes 3 cycles and every other cycle, another count can be produced through pipelining [74].

leading to a smaller fan-out. Checksums introduce a greater performance penalty than parity bits since all according data elements used to compute the checksum must be accessed to verify the checksum again. We add four checksums to each node: one for the metadata, i.e. parent node pointer, tree level and fill level (CS_M), one for all keys (CS_K), one for all values (CS_V) and one for all pointers (CS_P), as can be seen in Figure 4.17. This has a rather small impact on parameter K, as is depicted in Table 4.10: for 4-byte keys/values there remain 252 keys/values, instead of 254, the memory overhead is 24 bytes, and the fan-out decreases by 0.8%. For 8-byte keys and values, there are still 168 pairs (K=84), with an additional 32 bytes which fit into previously unused memory. All of the checksums must be updated whenever a respective node member is added, removed, or updated. During tree traversal, first CS_M is validated, whereas CS_K is validated while scanning through a node. Therefore, while comparing the node's keys with the searched one, they are XOR-ed to generate the validation checksum in parallel. Before descending or when the key is found all remaining keys are XOR-ed and the computed checksum is compared against the stored one. Also, only when the key is actually found in the node is the values checksum validated. Additionally, before descending, CS_P is validated by XOR-ing all child pointers.

EDB-Tree With Parity Bits and Checksums

We can also combine the previous two techniques of parity bits per key and value and checksums. By that, we potentially increase the bit flip detection capabilities, since a bit flip now only stays undetected when both codes fail. The node layout is then still the same as for the EDB-TreeCS and the parity bit is set again in the MSB, as for the EDB-TreePB. We call this variant *EDB-TreePBCS*.

EDB-Tree With AN Coding

For the EDB-Tree, we only considered XOR checksums so far in the form of checksum words and parity bits. Of course, the combination with AN coding is in principle possible, but in this thesis we only considered XOR checksums. We leave the detailed investigation, which should be straight forward, and performance evaluation of AN coded EDB-Trees to future work.

Triple Modular Redundant B-Tree

As a second baseline for the throughput measurements following later in Section 6.4, we use a TMR variant of the baseline B-Tree. It incorporates three sub-B-Trees and delegates all function calls to the three internally managed B-Tree instances. Likewise, for querying operations like key lookup, all three sub-trees are queried and afterwards, the majority is voted. When at least two out of the three results are the same, then this is assumed to be the correct result.

4.4 SUMMARY

In this chapter, we investigated the hardening of the storage layer of modern main memory-centric column stores using our two selected coding schemes AN coding and XOR checksums. Therefore, we first revised individual parts of that storage layer and, in more detail, the internal layer containing the physical schemas. These schemas deal with various aspects. First, they define the storage model which can in principal be row-based (NSM) or column-based (DSM), where column stores use the DSM-based layout. Considerations apply to mixed layouts like PAX respectively. Second, the data representation dictates the mapping from the various logical data types to some physical implementation data types, on the one hand, where for the sake of correctness, floating-point numbers are avoided in favor of fixed-point decimal arithmetic. On the other hand, aspects like lightweight integer compression are also considered. There are logical techniques, e.g. DICT, RLE, FOR, and DELTA, and the physical one NS. Actual lightweight compression algorithms are compositions of these techniques. Third, the data layout of the actual bits and bytes can be horizontal, vertical, or also a mixture like ByteSlice. Fourth and finally, next to the base data there are index structures as secondary access paths to speed up data lookup. Here, the B-Tree and various variants are also used in main memory-centric column stores.

As we showed, to harden the storage layer we only need to adapt that lowest internal layer, while the conceptual and external layers stay unchanged. Since we only deal with integers in the base columns, we harden them all using AN coding. We then discussed the problems of variable-width, limb-based representations of, e.g., decimal and string values. For decimal numbers each limb must be hardened individually, as a technical limitation today, because we did not yet obatin golden As for data widths over 32 bits. When golden As for larger data widths are computed, the first approach can be tested for feasibility, which we leave to future work. For string values, we showed that limb-encoding is also possible and compared this against using XOR checksums. XOR is faster than AN coding for string comparison and can be used in favor of efficiency ($\mathcal{R}2$). When effectiveness is of greater concern, then AN coding is the better choice. Next, we discussed the combination of hardening and lightweight integer compression. While it is quite natural to combine both, we now first must determine the data bit width which results from the logical compression to select an appropriate golden A. This will result in a 2-pass algorithm for hardened compression and we gave a concrete example for hardened 4-wise byte null suppression. Regarding the physical layout of hardened data, we discussed that the vertical and hybrid (ByteSlice) layouts are problematic due to high tuple reconstruction costs, since they distribute the bits or bytes of a value across many machine words. Consequently, we only use the horizontal data layout. Furthermore, we explained that UDI operations are orthogonal to the low-level data layout techniques.

Finally, we showed how to harden the B-Tree index structure against bit flips, as it is still one of the widely used secondary access paths. This resulted in the new EDB-Tree index data structure. We exploit virtual memory management to logically place all nodes in a distinct area in virtual memory space, which allows more pointer sanity checks than there were in literature. Furthermore, we investigated the use of parity bits and checksums, to harden meta data, keys, values, and pointers. Bit flip detection is done in each and every tree traversal, for each and every node.

The methods presented in this chapter provide a principal toolbox for building a *storage layer*, which is hardened against bit flips, for main memory-centric column stores. As we can see, detailed knowledge about our system leads to the use of different hardening techniques, depending on which kind of data must be protected. This is in accordance with Requirement $\mathcal{R}5$ – *separation of concerns*. Being equipped with a hardened storage layer, we still need to figure out two aspects. First, we need to investigate how this affects query processing, which is what we will show in the following. Second, we must also investigate the runtime implications in detail, which we will provide after query processing.

5

BIT FLIP DETECTING QUERY PROCESSING

Now that we are equipped with a hardened storage layer, we will adapt the query processing part. Two questions arise: Firstly, *when* to actually do bit flip detection and, secondly, in how far database operators must be adapted for this task. This is a crucial point, because this will impact the performances of a DMS in terms of availability, reliability, query throughput, and query latency. There are basically two possibilities: *online* and *offline* error detection. The former employs error detection during query processing, i.e. it is part of a query itself, while the latter does error detection as a completely separate operation. Based on our thesis error model (Definition 4) and Requirements $\mathcal{R}1$ and $\mathcal{R}4$, respectively effectiveness and availability, in this thesis we consider *online* error detection. Then, values can be checked for bit flips as soon as they are fetched from CPU cache or main memory, and this guarantees better availability since it is no separate process. We leave the consideration of offline detection to future work. Based on that, detecting bit flips too infrequently may allow them to stay silent and corrupt query results, and when error detection requires too much time or too many resources, it might influence the throughput or latency of queries too much. We will solve these issues with the help of the following main contributions:

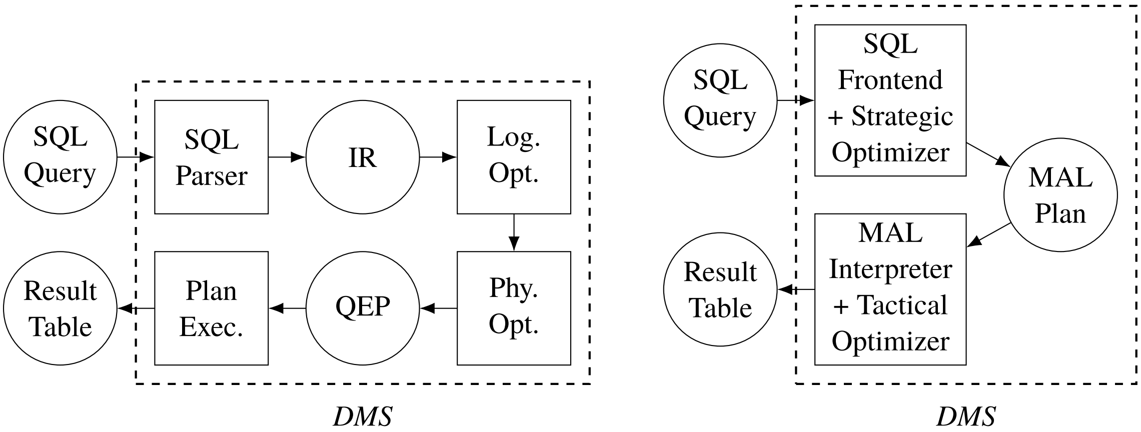
1. We introduce and discuss several *opportunities* for employing online error detection during query processing. This results in holistic, *continuous* bit flip detection, which was published in [87].
2. We show how to tightly integrate error detection primitives into physical query operators, also published in [87].
3. As a consequence of the very close integration, we show that this does *not* affect other parts of the query processing life cycle, except the physical operators' implementation.
4. We investigate the hardening of query intermediate results, in particular hardening bitmaps and the materialization of results.

To the best of our knowledge, we are the first to introduce arbitrary bit flip detection into query processing. This chapter is structured the following. First, we shortly revise query processing in column stores in Section 5.1. We need this to afterwards describe our hardened query processing approach in Section 5.2. This one, in turn, does error detection during query processing in an *online* fashion. Then, we also discuss the format and the handling of hardened intermediate results in Section 5.3. Section 5.4 summarizes and concludes this chapter.

5.1 COLUMN STORE QUERY PROCESSING

The reliable storage is only one part of data management, the other being the reliable query processing. In this section, we will first very briefly revise the basic steps of actually querying data from a DMS. We will then compare this to modern systems tailored for in-memory processing, i.e. concepts which are specific to in-memory column stores. This provides an overview of the concepts we need afterwards, whereas the query life cycle is discussed in great detail by many standard text books, like [159], and research papers, e.g. [21]. Based on this overview, in the following section we explore opportunities for integrating bit flip detection into query processing.

Each DMS offers some way to work on the managed data through queries in a specific language, which for relational databases is typically some SQL dialect. The way to get data out of (or into) the database is comprised of a whole *query life cycle*, which is depicted in a general way in Figure 5.1a.



(a) Old-fashioned query life cycle, after [159]. (b) MonetDB-style query life cycle, after [21].

Figure 5.1: The coarse query life cycle.

The abstract concepts are represented in rectangles and circles. First, a user formulates a *query* in, e.g., SQL. Systems have also been developed to provide multiple language front-ends, e.g. MonetDB additionally provides an object-oriented front-end [21]. A (SQL) parser translates this query into a DMS-specific internal representation (IR). In Figure 5.1, (by-) products like a query are drawn as circles, while components, such as the parser, which take inputs and produce outputs are drawn as rectangles. Conceptually, the IR is a tree of *logical operators* representing the basic algorithms and their order for operations on the managed data. In the relational algebra, these operators are among others selection (σ), projection (π), join ($\bowtie$), and grouping (γ). The first one, σ, selects tuples from a relation according to a (simple or complex) predicate. The projection π is used to select from a table only a subset of the attributes (columns). The join operator, $\bowtie$, builds (a subset of) the Cartesian product of two tables according to a join attribute [117]. The operator γ is used to group attributes with reference to one other attribute, which is often used in conjunction with aggregating the non-grouped attributes, e.g. computing the minimum, maximum, sum, average, and the like. An example query tree is shown in Figure 5.2a, which we will cover in detail later. The internal representation is then given to two optimizers, which try to reduce the total query costs. First, the *logical optimizer* does optimizations on the algebraic level, which for instance includes reordering the operators. In particular, the ordering of join operators is a very important optimization problem [117], but also selection-push down is a well-known optimization and there are many others [159]. Afterwards, the *physical optimizer* maps the logical operators to physical operators, where it may typically choose from a set of available implementations. This depends on data characteristics like sorting and selectivities. For instance, a concrete join implementation on completely unsorted data may be based on nested loops or hashing, while when one or both join inputs are sorted, a sort-merge join might be preferred [117, 159]. The selection (σ) can, for instance, be realized through full column scans, or index scans. It is important to note, here, that physical optimization is typically done *before* query execution, based on estimated cardinalities. The result of the physical optimization is a query execution plan (QEP), which is the final "recipe" which is then executed. In Volcano-style database systems [61], the query plan consists of tree-like connected operators, where upper physical operators call the lower physical operators in the execution plan to fetch individual tuples for processing. This is known as tuple at a time execution. At the leafs there are typically the tables from which the business data is fetched. After the execution of the query plan, the result is returned to the user, which in the case of relational database systems is a result table.

We call the above the "old-fashioned" style, because in the recent past some aspects of this life cycle were shown to be suboptimal in the context of in-memory DMSs. For instance, MonetDB [21] uses a different approach, shown in Figure 5.1b. On a first glance, it seems like less steps are involved, but that stems from the fact that parsing, IR, and optimization steps are merged into the SQL frontend. Furthermore, there are no logical and physical optimizations, but *strategic* and *tactical* optimizations, which are *not* synonyms. The former combines logical and parts of physical optimization. Instead of a tree-like query representation, a sequential MonetDB assembly language (MAL) plan is generated, where the *order* of the operators is fixed, but not the choice of physical implementations. That choice is made during the latter tactical optimization at run-time, by the MAL interpreter. This can alleviate the problem of making decisions based on wrong estimates [75]. For each logical operator, there are specializations depending on, e.g., the input data types, sorting denseness etc. of the inputs. A detailed description of MAL is out of scope of this thesis and is described in full length in [21]. We will give a small example just below.

Another major difference is the execution model: in contrast to the pull-based tuple at a time execution, where complete tuples are passed between operators, MonetDB and virtually any in-memory column store use an operator (or column) at a time execution model. Here, DSM comes into play (cf. Section 4.1.2) where each column is stored separately. By that, the MAL plans can be sequential recipes and each operator processes a whole column. MAL also allows parallel sections to work on partitions of columns in separate threads. The required partitioning is done dynamically at run-time by calculating the array bounds which each thread should process on its own, which is known as vector at a time execution model. The benefit is that it allows easy partitioning at run-time into chunks (vectors) and straight-forward parallelization. In MonetDB, the actual data model uses so-called binary association tables (BATs), which actually consist of two columns, as pointed out in Section 4.1.2 in Figure 4.3c. While we used the term "column" so far, in MonetDB we use the notion of BATs, however the expressiveness is the *very same*. For the current high level view it suffices to think of BATs as single columns[1]. The tactical optimization now chooses physical operators based on the input BATs' data types, data characteristics, sorting, available secondary access paths (indexes) and more [21]. Like before, the result of a query is a relational table for relational SQL queries.

In the following, we want to first describe in more detail some peculiarities of MAL plans and then present our adaptations for hardening query processing. For that, we use the following example query, which is based on the SSB schema [131]:

```sql
SELECT p_name, SUM(lo_revenue) FROM lineorder JOIN part ON lo_partkey = p_partkey
    WHERE p_color = 'goldenrod' and lo_orderdate = 19971010 group by p_name;
```

Listing 5.1: Concrete example query.

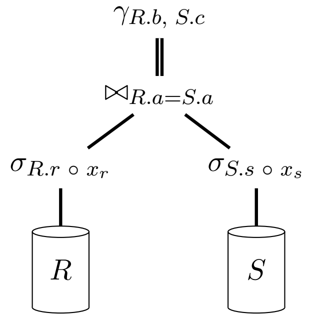
In a nutshell, this query selects the name of parts with color "goldenrod", which were sold on October 10, 1997. In addition, for each part the total revenue is summed up. To make the upcoming graphics more concise, we will use the following abstract form of the above query:

```sql
SELECT R.b, SUM(S.c) FROM R JOIN S ON R.a = S.a WHERE R.r ∘ x_r AND S.s ∘ x_s GROUP BY
    R.b
```

Listing 5.2: Abstract example query.

This translates into a relational query plan, shown in Figure 5.2a. There are two tables `t1` and `t2` from which we query one column each, R and S, respectively. The projection operator π, which selects the two columns from the tables, is omitted here and typically not needed anyways due to the

[1] MonetDB already uses a so-called headless mode, which means it physically really only single columns are passed between operators [14].



(a) Relational Query plan.

```
1  L_O      :bat[int]  :=  sql.bind ("ssb", "lineorder", "lo_orderdate");
2  L_OID    :bat[oid]  :=  sql.tid ("ssb", "lineorder");
3  SEL_LO   :bat[oid]  :=  algebra.thetaselect (L_O, L_ID, 19971010, "==");
4  L_P      :bat[int]  :=  sql.bind ("ssb", "lineorder", "lo_partkey");
5  PROJ_LP  :bat[int]  :=  algebra.projection (SEL_LO, L_P);
6  P_C      :bat[str]  :=  sql.bind ("ssb", "part", "p_color");
7  P_OID    :bat[oid]  :=  sql.tid ("ssb", "part");
8  SEL_PC   :bat[oid]  :=  algebra.thetaselect (P_C, P_ID, "goldenrod", "==");
9  P_P      :bat[int]  :=  sql.bind ("ssb", "part", "p_partkey");
10 PROJ_PP  :bat[int]  :=  algebra.projection (SEL_PC, P_P);
11 (JOIN_L  :bat[oid], JOIN_P :bat[oid]) := algebra.join (SEL_LP, P_ID);
12 L_R      :bat[int]  :=  sql.bind ("ssb", "lineorder", "lo_revenue");
13 PROJ_LR  :bat[int]  :=  algebra.projectionpath (JOIN_L, SEL_LO, L_R);
14 P_N      :bat[str]  :=  sql.bind ("ssb", "part", "p_name", 0);
15 PROJ_PN  :bat[str]  :=  algebra.projectionpath (JOIN_P, SEL_PC, P_N);
16 (PN_GRP  :bat[oid], GRP_PN :bat[oid]) := group.groupdone (SEL_PN);
17 RES_PN   :bat[str]  :=  algebra.projection (GRP_PN, PROJ_PN);
18 RES_LR   :bat[hge]  :=  aggr.subsum (PROJ_LR, PN_GRP, GRP_PN);
```

(b) MAL plan.

Figure 5.2: Unprotected processing.

nature of column stores. The base tables are indicated as cylinders and they are filtered on equality predicates $R.r = x_r$ and $S.s = x_s$ with some arbitrary arguments x_r and x_s, from columns $R.r$ and $S.s$, respectively. Afterwards, both tables are joined on columns $R.a$ and $S.a$. The single lines between the base columns, the filters σ and the join operator indicate that the intermediate results are again just columns with single payload. The join produces an intermediate column which contains *pairs* (r, s), which is indicated by the double line. Finally, the group operator (γ) groups this intermediate on attribute r and we assume that it also produces the sums over the groups of values s. In a real DMS, this might result in two or more actual operators, but for conciseness we use a single operator. We will later describe the integration of error detection primitives in a few physical operators. Therefore, we now describe two examples for the unencoded case, namely the filter (selection, σ) and join ($\bowtie$) as examples for single-input and multi-input operators, respectively. We assume that the plan shown in Figure 5.2a is an executable query plan, where the physical optimizer already selected some concrete operator implementations. For instance, the selection operators might be full column scans, while the join could be a nested loop join and the grouping operator could be hash-based.

In contrast to the relational plan, Figure 5.2b shows a MAL plan, which we pruned for better readability. That plan represents the first query variant with concrete schema, table, and column names. A graphical representation of that plan is provided in Figure 5.3. In the MAL plan, each line represents a single operator, which may return a single BAT, or even multiple ones put in between parentheses (e.g. on lines 11 and 16). Keywords are denoted in **bold** font, while strings

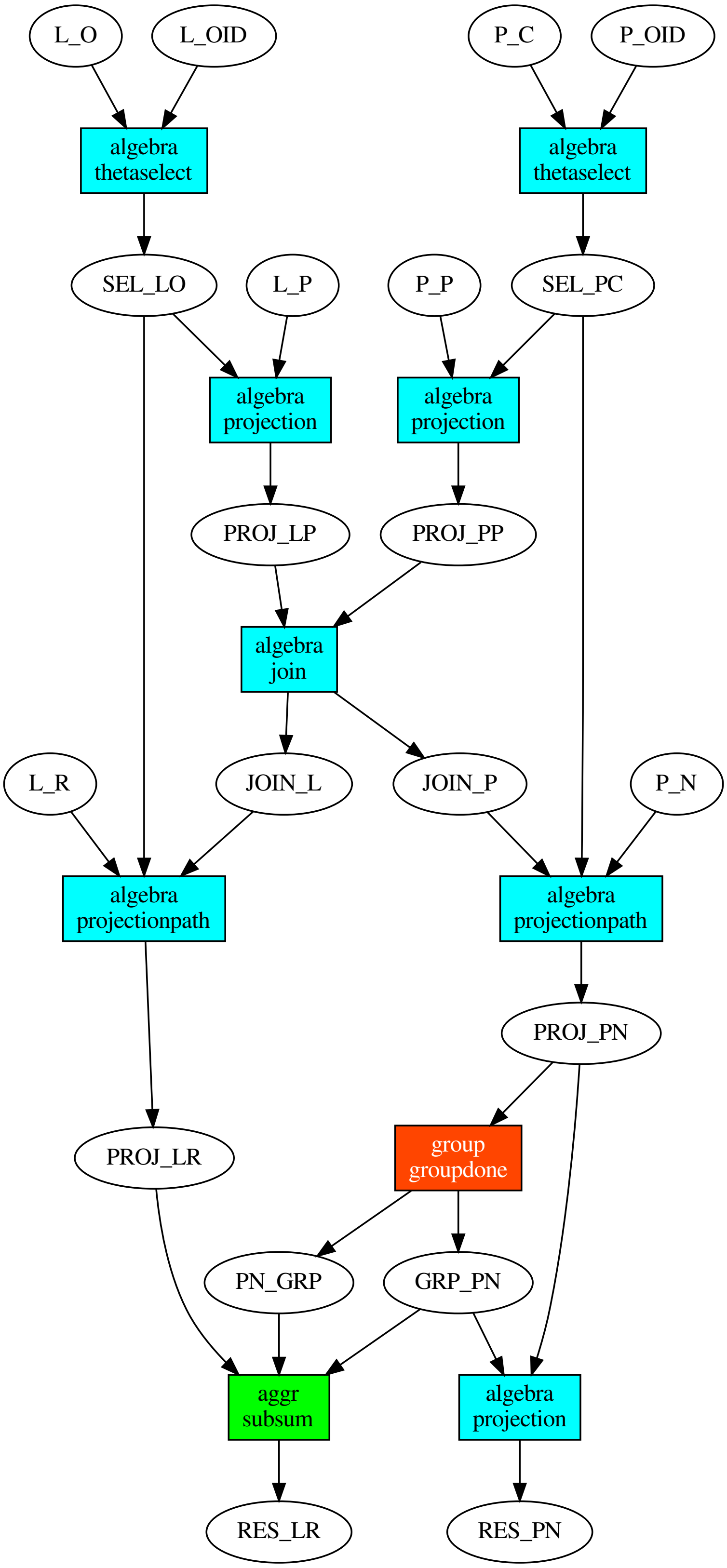


Figure 5.3: MAL plan graph

are shown *slanted*. The operator result is *assigned* to one ore more BATs, denoted by `:=`. Further, MAL is a *typed* language and for each BAT the type is explicitly given as `:bat[<type>]`, where `<type>` in this case is e.g. `int` for integers, `oid` for OIDs or VOIDs, and `str` for strings. As a side note, the operators are grouped into several modules, e.g. `sql`, `algebra`, or `bat`. In the MAL plan, an operator call is preceded by its module and has the form `<module>.<operator>`. In Figure 5.3, the operators are shown in colored boxes with the module on the first line of each label and the operator name on the second line. According to the query, first columns `lo_orderdate` and `p_color` are filtered (lines 1–3 and 6–8, respectively). The BATs are here given more descriptive abbreviations to resemble the respective table and column, or additionally the operator which produced them. For instance, `L_O` is base column Lineorder.lo_Orderdate (line 1), while `SEL_LO` is the result from the selection on the `lo_orderdate` column using predicate `lo_orderdate` == 19971010+ (line 3). Additionally, a few BATs are explicitly used to denote the OIDs of the base columns, e.g. `L_OID` (line 2). While these must be given to the `thetaselect` operators explicitly, this allows to do cascaded selections on multiple columns of the same table, using e.g. a select-project-select pattern. A `projection` operator in a MAL plan takes two input BATs, one column BAT and one OID BAT and fetches from the column those values which correspond to the given OIDs. Consequently, a `projection` operator is *not* a projection in the classical sense. The `projection`s before the MAL join (lines 4–5 and 9–10) are required, because the initial selections are done on columns different from those which are joined. The `thetaselect`s return OID BATs which only contain the column *positions* of the matching values. The `projection`s then fetch only those values from the to-be-joined columns which correspond to the tuples previously selected (having the same OID). Consequently, the join only operates on these "projections" (line 11). The join operator returns two BATs, where each BAT contains the OIDs pointing into the respective input BATs. Since the final result contains columns which are even different from the selection and join columns, two more `projection`s are necessary (lines 12–15). Then, the grouping is performed (line 16), where two important BATs are returned: The first one `PN_GRP` is a mapping from the input positions to the groups, denoted as OIDs which point into the second BAT `GRP_PN`. That second one contains as many OIDs as there are distinct groups and its OIDs point to the first occurrence (position) in the input BAT, so that the actual value denoting that group can later be retrieved through a `projection` again (line 17). Consequently, the same methodology as for the `thetaselect` is used here. Finally, the sums over the revenues with respect to the `p_color` groups are computed (line 18). The BATs which are finally printed as the result are `RES_PN` and `RES_LR`, which contain the grouped part names and revenue sums per part, respectively.

The relational plan from Figure 5.2a and the MAL plan from Figures 5.2b and 5.3 can be loosely matched, which shows that both relational query plan and MAL plan are still closely related. The two selection operators from Figure 5.2a are represented in Figure 5.3 by a combination of `algebra.thetaselect` followed by a `algebra.projection`, each. The intermediate results from the `thetaselect`s are reused later for projecting the revenue and name columns, `L_R`→`PROJ_LR` and `P_NPROJ_PN`, respectively. The `join` is given in both plans. Finally, the grouping and sum operator from Figure 5.2a are represented in the MAL graph by two `projectionpath`s, a `groupdone`, a `subsum`, and a `projection`. Figures 5.2b and 5.3 show that the MAL plan as well as its graphical counterpart are rather complex. Consequently, as we showed that both can still be matched to relational query plans (and its graphical counterpart), we will later only use the much more concise relational query graphs to introduce some new concepts. Before we do so, however, we will show for two concrete operators how they work, so that we can later also show on the operator level how to integrate bit flip detection.

Algorithm 5.1 lists a filter operator, which takes as input a BAT B_{in}, a comparison operator (which is typically hard-coded), and the filter predicate *pred*. When a range predicate is filtered, two predicates are provided for the lower and upper bound, but otherwise the concept is the same. The filter operator simply iterates over all head and tail value pairs (h, t) (Line 1), evaluates the

Algorithm 5.1 Filter scan for an unencoded query.

```
Input: $B_{\text{in}}$                                  ▷ input BAT
Input: $\circ \in \{<, \leq, =, \neq, \geq, >\}$        ▷ comparison operator
Input: pred                                             ▷ filter predicate
Output: $B_{\text{out}}$                                ▷ result BAT
1: for each $(h, t) \in B_{\text{in}}$ do
2:     if t ∘ pred then
3:         append $(h, t)$ to $B_{\text{out}}$
4:     end if
5: end for
```

Algorithm 5.2 Nested loop join for an unencoded query.

```
Input: $B_{\text{in},1}$, $B_{\text{in},2}$             ▷ input BATs
Output: $B_{\text{out}}$                                ▷ result BAT
1: for each $(h_1, t_1) \in B_{\text{in},1}$ do
2:     for each $(h_2, t_2) \in B_{\text{in},2}$ do
3:         if $t_1 = h_2$ then
4:             append $(h_1, t_2)$ to $B_{\text{out}}$
5:         end if
6:     end for
7: end for
```

predicate for the tail payload (Line 2) and, when it matches, appends the pair to the output column B_{out} (Line 3). Especially filter operators can be vectorized [195] and, in particular, for the SSE and AVX ISAs all required comparison operators exist as vector instructions. In this case, the basic algorithm stays the same, except that the predicate is distributed into a vector and then whole vectors are compared.

Algorithm 5.2 lists a join operator using a nested loop algorithm, where an outer loop iterates of the first input $B_{\text{in},1}$ (Line 1) and for each of its pairs (h_1, t_1) iterates over the second input $B_{\text{in},2}$ in an inner loop (Line 2). For each pair of pairs the inner loop compares the tail of the first column t_1 with the head of the second column h_2 (Line 3) and when they match the first head h_1 and second tail t_2 are written to the output column B_{out} (Line 4).

To summarize this section, we briefly presented the important parts of the query processing life cycle. What is important for our case is that a query is parsed by some language-specific front end and the resulting internal representation is then optimized in two (or more) stages. While contemporary systems use logical and physical optimizations, in-memory column stores like MonetDB use strategic and tactical optimization. The actual query evaluation is done using a QEP. Modern systems like MonetDB generate a recipe which is then interpreted and operators are called on whole columns, resulting in the column at a time execution model. MAL is used as an internal representation in MonetDB and it also allows vector at a time execution by splitting up the columns through online range partitioning. In MAL, all columns, both base columns and intermediates, are realized in the form of BATs. We discussed for a concrete query its respective relational query plan and MAL plan. We furthermore showed that we can match the operators from the relational plan to (sets of) operators in the MAL plan. Due to that, we will only use the more concise relational representation in the following. Furthermore, there is a multitude of database operators and we described two physical operators in more detail. We will use these in the following to describe how to integrate bit flip detection primitives into operators.

5.2 BIT FLIP DETECTION OPPORTUNITIES

In this thesis we deal with the detection of bit flips in business data, which means the base columns, intermediate columns, and index structures. We will now discuss which parts of the query execution we need to adapt for hardened data in order to achieve holistic online bit flip detection. Basically, the query life cycle stays the same as before. The parser and the optimizers stay oblivious to the actual data hardening, so that we do not interfere with generating an optimal query plan. This greatly simplifies the integration of data hardening into query processing. We can use the optimized QEP as is and slightly adapt it to our needs. We assume that all base columns contain integers hardened with AN codes, only. Based on that, there are several *detection opportunities*, which exhibit various advantages and disadvantages that we will explain in the following. Since we only deal with bit flip *detection* in this thesis, we leave the concrete error handling to future work.

5.2.1 Early Onetime Detection

We call our first variant *Early Onetime Detection*, which is a very *unintrusive* way of combining data hardening and query processing. The idea is to introduce a new physical database operator Δ with the sole purpose of performing a decode-and-detect run on a single column. Based on the query plan from the optimizers (Figure 5.2), we insert the Δ operator before any other operator, even before column scans (σ). This does not interfere with the original query plan as the data characteristics are not changed. The new operator is shown in Figure 5.4, where the encoded data is visualized using blue dotted lines. Columns R and S are both encoded using the same parameter A, however they could also be encoded with different As. The Δ operator simply performs a single scan over hardened data and performs appropriate bit flip detection and writes as output the *decoded* values, as is indicated by solid black lines, again. The greatest advantage of this approach is, that for all of the *existing* physical operators we need not care about the actually employed error control code. Only the Δ operators need to take care of that. Consequently, no operators need to be adapted, which makes the integration of bit flip detection quite easy. We only insert the Δ operator after the query optimizer returns the optimal plan. There are, however, some disadvantages. First, there is only a single point of error detection when data is read the first time from a base column. Second, complete base columns are re-materialized in unencoded form. On the one hand, this can lead to a huge memory demand per query, as all required base data is stored a second time. On the other hand, by that the remaining, original part of the query works on unencoded data and bit flips cannot be detected any longer.

As we leave error correction to future work, here the Δ operator only collects hardened array positions of the input columns. These positions represent those values at which a bit flip was detected and are stored in another intermediate array. This abstraction allows to plug in concrete error handling techniques in future work. Consequently, next to the unencoded base data, Δ also returns this array of hardened positions to corrupt values. If no other error handling is conducted, the latter can simply be tested to be empty and otherwise be ignored. By that, this does not interfere with the remaining processing.

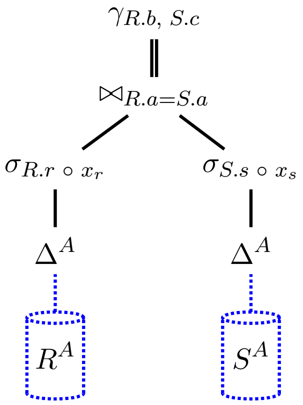
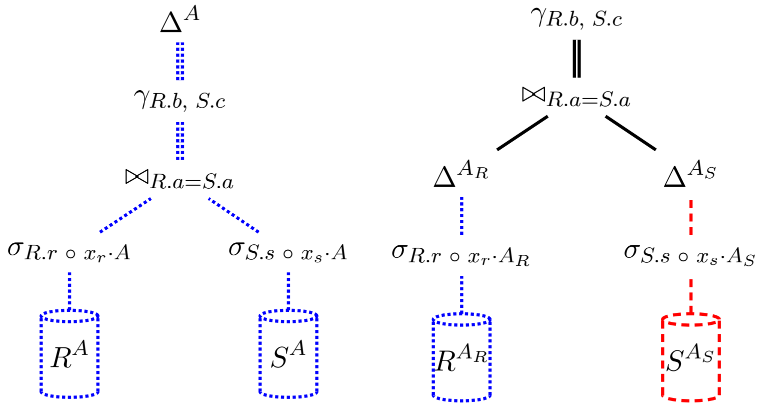


Figure 5.4: Query plan for Early Onetime Detection.



(a) Same As.

(b) Different As.

Figure 5.5: Query plan for Late Onetime Detection.

5.2.2 Late Onetime Detection

The next variant is *Late Onetime Detection*, which inherits the detect-and-decode Δ operator from before. However, we now exploit the fact that we use AN coding in all our base columns and that it is an *arithmetic* code which allows to directly work on hardened data. Figure 5.5a shows how this allows to move the Δ operator to the top of the query and the blue dotted lines run from bottom to top. This means, we need not decode the data during the part of the original query. Using this approach, we *must* take care of some of the operators' parameters. For scan operators, we then need to encode the predicates, as well, so that both values and predicates are encoded with the same A. This is shown in Figure 5.5a for the σ operators, where both filter predicates are encoded as $x_r \cdot A$ and $x_s \cdot A$. For multi-input operators like column join, we must distinguish two cases. First, when all inputs are encoded with the exact same A, then no action is required. For instance, a join on 2 input columns (or intermediates) C_1 and C_2 can leverage the fact that

$$c_1 = c_2 \Leftrightarrow (d_1 \cdot A) = (d_2 \cdot A) \Leftrightarrow d_1 = d_2 \quad \text{, with } c_1 \in C_1, c_2 \in C_2.$$

This is the case actually shown in Figure 5.5a. Second, when inputs are encoded with different As, then the Δ operator must precede any multi-input operators with differently encoded inputs. This is required, because such operators would generate only false positives, since

$$(d_1 \cdot A_1) = (d_2 \cdot A_2) \overset{A_1 \neq A_2}{\Rightarrow} d_1 \neq d_2$$

for two input columns C_1 and C_2 which are encoded with different parameters $A_1 \neq A_2$. This case is shown in Figure 5.5b, where both columns R and S are encoded with different parameters A_R and A_S, respectively. The predicates for the filter operators are encoded, again, appropriately to match their input column code parameter. Afterwards, Δ hands over unencoded intermediates to the join operator and the rest of the query works on unencoded data. One advantage of Late Onetime Detection is, that, it is still a very unintrusive approach which, like Early Onetime Detection, only requires the additional Δ operator. Since this allows us to move the Δ operator to the *end* of the query, we perform error detection on potentially less hardened values than for Early Onetime Detection. Of course, when the Δ operator is located after join operators which generate more join pairs than input values, then it can also be the other way around. However, as for Early Onetime, a disadvantage is still the single point of error detection for each hardened value. Also, errors *prior* to the invocation of the Δ operator are not detected, in contrast to Early Onetime. Then, determining whether intermediate results are corrupted, or even base date, requires additional action in the form

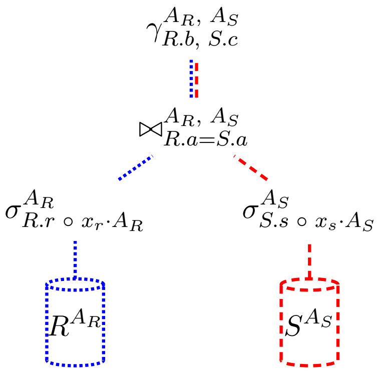

$\gamma^{A_R,\, A_S}_{R.b,\, S.c}$
$\bowtie^{A_R,\, A_S}_{R.a=S.a}$
$\sigma^{A_R}_{R.r \circ x_r \cdot A_R}$
$\sigma^{A_S}_{S.s \circ x_s \cdot A_S}$
R^{A_R}
S^{A_S}

Figure 5.6: Query plan for Continuous Detection.

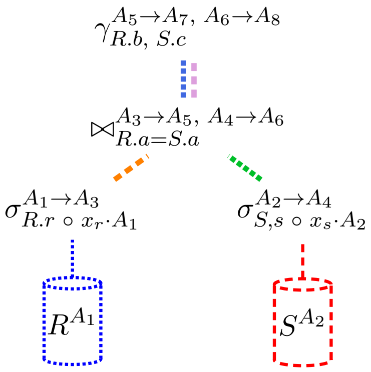

$\gamma^{A_5 \rightarrow A_7,\, A_6 \rightarrow A_8}_{R.b,\, S.c}$
$\bowtie^{A_3 \rightarrow A_5,\, A_4 \rightarrow A_6}_{R.a=S.a}$
$\sigma^{A_1 \rightarrow A_3}_{R.r \circ x_r \cdot A_1}$
$\sigma^{A_2 \rightarrow A_4}_{S,s \circ x_s \cdot A_2}$
R^{A_1}
S^{A_2}

Figure 5.7: Query plan for Continuous Detection and recoding in each operator.

of running an error detection sweep on both. Arithmetic operations must be handled as indicated in Equations (3.14) and (3.16) to (3.21). For instance, addition and subtraction must be done either on unencoded data or on data hardened with the same parameter. The multiplication requires one of the operators to be unencoded, and the division must first divide two encoded operands and then encode again. Consequently, for division and multiplication, either these operators must be adapted to take care of the hardened data, or the Δ operator must be put before them in the query plan. This affects also σ filter operators, which evaluate predicates containing such arithmetic operations.

5.2.3 Continuous Detection

The *effective* detection of bit flips is the major goal of this thesis and the previous two methods, Early and Late Onetime Detection, tackle this issue only to a limited degree. To reach our goal, we need a *holistic* approach, which we call *Continuous Detection*. In contrast to the previous approaches, it requires to adapt *all* physical query processing operators. The idea is to perform error detection in *each and every* operator, on *each and every* hardened value which is read, both from main memory and from the CPU caches. This approach does not require the Δ operator and does not inherit its limitations from decoding data. This also alleviates us from the task of choosing positions for Δ operators. As a side effect, we are even indifferent to the fact whether a hardened value was corrupted already in main memory, during data transmission between main memory and CPU, or even in one of the CPU caches. The disadvantage of this approach is the *engineering effort* for adapting all physical query operators to contain error detection primitives. However, in contrast to the previous approaches, integrating error detection primitives into the physical operators also allows *recoding* of hardened data *on the fly*. Again, this is possible for each and every value. Therefore, we use Equation (3.63), which is a *constant* factor that needs to be computed only once in an operator.

Figure 5.6 shows the query plan when employing the Continuous Detection approach. The Δ operator is not needed any longer and, as in Figure 5.5b, we can basically assume that all columns are potentially hardened with different As. The superscripts at the operators highlight that they are AN coding-*aware* and contain detection primitives. We will also call them “hardened operators” in the following. Also, the filter predicates are encoded using the appropriate columns’ As. As we can also see in Figure 5.6, even the join operator emits the intermediate results with differently hardened values, depending on the input columns.

A query containing data recoding is shown in Figure 5.7, where each operator recodes data on the fly, indicated by different colors. The operators denote by a superscript the recoding from a source parameter A_{src} to a destination parameter A_{dst} as $A_{\text{src}} \to A_{\text{dst}}$. The constant factor for this transformation is computed as in Equation (3.63). Hardening or softening during query processing can be used to influence the memory size of intermediate results. For instance, let us for now assume that we are given an error model, where for an intermediate result it suffices to detect less bit flips than for the base data, due to the much shorter temporal residence in main memory. Then, we might pick a smaller A from Table 3.6 or Tables A.1 to A.4 for the intermediate results and recode the operators' outputs accordingly as in Figure 5.7. Another use case could be special queries to recode base data. In this case, we assume that there is some system component or service, which can predict the required detectable bit flip weight for the base data. This would be required to adapt to hardware aging effects. Then, a special query could increase the data hardening in an online fashion and following queries would use that reinforced data. The driving idea behind such an approach is to satisfy Requirement $\mathcal{R}4$, availability, to not stall queries on the to be recoded data.

We will now discuss how to adapt physical query operators. Algorithm 5.3 lists the AN coding-aware variant of a filter scan, for which we presented the unencoded one in Algorithm 5.1. Here, we see first that there are several more arguments which the filter operator takes, where all encoded columns and values are marked by *. In contrast to Algorithm 5.1, the input column B^*_{in} and the predicate *pred** are AN encoded and the operator needs to know the head's and tail's AN code parameters $A_h, A_h^{-1}, A_t, A_t^{-1}, A_{\text{pos}}, d_{h,\min}, d_{h,\max}, d_{t,\min}, d_{t,\max}$. The encoded filter operator returns not only an output column containing hardened values B^*_{out}, but also *position vectors* v_h, v_t which contain the (hardened) positions of corrupt values in the head and tail, respectively. This is the same behavior as for the Δ operator and, as before, we use this model because we do not consider error correction in this thesis. However, this provides an interface for future work, where error handling strategies can be plugged in. The positions are themselves AN hardened using the given A_{pos}. If recoding is desired, then parameters A'_h and A'_t must be provided, too. Algorithm 5.3 lists the operator variant which also recodes its input. In the operator, first the variable to keep track of the position is initialized to zero (Line 1) and storage for the encoded position vectors is allocated (Lines 2 and 3). Then, the **const**ant recoding factors are computed on Lines 4 and 5. When recoding is not desired, the appropriate operations can simply be left out. The loop on Line 6 iterates over the hardened input column and we use a temporary variable `valid` to track whether both head and tail values are valid (Line 7), which we initialize to the Boolean true value $\top$. We then first test the head for detectable bit flips on Lines 8 and 9 using Equation (3.68) and Equation (3.67) and if a corruption was detected, the encoded position is appended to the head position vector (Line 10) and `valid` is set to Boolean false $\perp$ (Line 11). For unsigned integers, we only need the latter comparison. The same goes for the tail on Lines 13 to 16. If both values are valid, the filter predicate is evaluated (Line 18) and the recoded values are appended to the output column (Line 19). By that, we also make sure to always check both head and tail for errors. When the head is of type VOID, the respective checks can be omitted, since this type represents purely virtual data. Finally, the position tracking variable `pos` is incremented on Line 21. We assume that the variables `pos` and `valid` are register or stack variables, which for the runtime of the operator are not corrupted.

Algorithm 5.4 shows the encoded counterpart of the join operator from Algorithm 5.2. The additions are essentially the same as in Algorithm 5.3, with additional parameters for all inputs and we now have two sets of position vectors. These are initialized on Lines 1 and 2. The recode factors are now computed for the first input's head and second input's tail on Lines 4 and 5, respectively. Furthermore, all inputs must be checked for bit flips, first in the outer loop (Lines 6 to 16) and

Algorithm 5.3 Filter scan for Continuous Detection, having head and tail hardened with different As and recoding its output. * denotes hardened values and columns. Subscript $_h$ and $_t$ denote head and tail, respectively.

```
Input: $B^*_{\text{in}}$                                            ▷ Encoded input column
Input: $A_h, A_h^{-1}, A_t, A_t^{-1}, A_{\text{pos}}$               ▷ AN coding parameters
Input: $A_{\text{pos}}$                                             ▷ Position vector code parameter
Input: $A'_h, A'_t$                                                 ▷ Recoding parameters
Input: $d_{h,\text{min}}, d_{h,\text{max}}, d_{t,\text{min}}, d_{t,\text{max}}$   ▷ Domain range for $B^*_{\text{in}}$
Input: $\circ \in \{<, \leq, =, \neq, \geq, >\}$                    ▷ Comparison operator
Input: pred*                                                        ▷ Predicate encoded with $A_t$
Output: $B^*_{\text{out}}$                                          ▷ Result BAT (same $A$ as $B^*_{\text{in}}$)
Output: $v^*_h, v^*_t$                                              ▷ Position vectors for head and tail
 1: pos ← 0
 2: $v^*_h.allocate()$
 3: $v^*_t.allocate()$
 4: const $A_{h,\text{recode}} \leftarrow A_h^{-1} \cdot A'_h$
 5: const $A_{t,\text{recode}} \leftarrow A_t^{-1} \cdot A'_t$
 6: for each (head*, tail*) ∈ $B^*_{\text{in}}$ do
 7:     valid ← ⊤
 8:     $h \leftarrow$ head* $\cdot A_h^{-1}$
 9:     if $h < d_{h,\text{min}}$ or $h > d_{h,\text{max}}$ then
10:         $v^*_h.append(A_{\text{pos}} \cdot pos)$
11:         valid ← ⊥
12:     end if
13:     $t \leftarrow$ tail* $\cdot A_t^{-1}$
14:     if $t < d_{t,\text{min}}$ or $t > d_{t,\text{max}}$ then
15:         $v^*_t.append(A_{\text{pos}} \cdot pos)$
16:         valid ← ⊥
17:     end if
18:     if valid = ⊤ & tail* ∘ pred* then
19:         append (head* $\cdot A_{h,\text{recode}}$, tail* $\cdot A_{t,\text{recode}}$) to $B^*_{\text{out}}$
20:     end if
21:     pos ← pos +1
22: end for
```

then in the inner loop (Lines 19 to 28). In the inner loop, we need not set the temporary Boolean variable `valid` again, because it must be true in order to enter the inner loop, anyways. We see here that the second input's values are checked multiple times, so we can stay oblivious to whether data was evicted from the CPU caches and must be freshly fetched from unreliable main memory. Finally, only if all values are valid, the first column's head and second column's tail are compared (Line 30) and on a match recoded and appended to the output (Line 31). While we only showed the nested loop join variant, other join implementations essentially require the same changes.

Algorithms 5.3 and 5.4 list several new parameters to the operators compared to the unencoded ones from Algorithms 5.1 and 5.2. However, the AN coding parameters and the data domain ranges for the input columns need not be real inputs. From the coding parameters, only the actual A needs to be stored in the columns' meta data. The inverse can be computed on the fly using the extended euclidean algorithm and the data domain ranges are implicitly given by the column data types. The parameter A_{pos} for encoding the values in the error position vectors can be statically defined for all operators, since this is always the same data type. Furthermore, the predicates which need

Algorithm 5.4 Nested loop join for Continuous Detection, having all inputs hardened with different As and recoding its output. * denotes hardened values and columns. Subscript $_h$ and $_t$ denote head and tail, respectively.

```
Input: $B^*_{\text{in},1}, B^*_{\text{in},2}$                                   ▷ Input columns
Input: $A_{h,1}, A^{-1}_{h,1}, A_{t,1}, A^{-1}_{t,1}$                           ▷ AN coding parameters for $B^*_{\text{in},1}$
Input: $A_{h,2}, A^{-1}_{h,2}, A_{t,2}, A^{-1}_{t,2}$                           ▷ AN coding parameters for $B^*_{\text{in},2}$
Input: $A_{\text{pos}}$                                                          ▷ Position vector code parameter
Input: $A'_h, A'_t$                                                              ▷ Recoding parameters
Input: $d_{h,1,\text{min}}, d_{t,1,\text{min}}, d_{h,1,\text{max}}, d_{t,1,\text{max}}$   ▷ Domain range for $B^*_{\text{in},1}$
Input: $d_{h,2,\text{min}}, d_{t,2,\text{min}}, d_{h,2,\text{max}}, d_{t,2,\text{max}}$   ▷ Domain range for $B^*_{\text{in},2}$
Output: $B^*_{\text{out}}$                                                       ▷ Result column
Output: $v^*_{h,1}, v^*_{t,1}, (v^*_{h,2}, v^*_{t,2}$                            ▷ Position vectors
for each $x \in \{(h,1),(t,1),(h,2),(t,2)\}$ do
    $v^*_x.allocate()$
end for
const $A_{h,\text{recode}} \leftarrow A^{-1}_{h,1} \cdot A'_{h,1}$
const $A_{t,\text{recode}} \leftarrow A^{-1}_{t,2} \cdot A'_{t,2}$
for each $(h^*_1, t^*_1) \in B^*_{\text{in},1}$ do
    valid $\leftarrow \top$
    $h \leftarrow h^*_1 \cdot A^{-1}_h$
    if $h < d_{h,\text{min}}$ or $h > d_{h,\text{max}}$ then
        $v^*_{h,1}.append(A_{\text{pos}} \cdot pos)$
        valid $\leftarrow \bot$
    end if
    $t \leftarrow$ tail$^* \cdot A^{-1}_t$
    if $t < d_{t,\text{min}}$ or $t > d_{t,\text{max}}$ then
        $v^*_{t,1}.append(A_{\text{pos}} \cdot pos)$
        valid $\leftarrow \bot$
    end if
    if valid = $\top$ then
        for each $(h^*_2, t^*_2) \in B^*_{\text{in},2}$ do
            $h \leftarrow$ head$^* \cdot A^{-1}_h$
            if $h < d_{h,\text{min}}$ or $h > d_{h,\text{max}}$ then
                $v^*_{h,2}.append(A_{\text{pos}} \cdot pos)$
                valid $\leftarrow \bot$
            end if
            $t \leftarrow$ tail$^* \cdot A^{-1}_t$
            if $t < d_{t,\text{min}}$ or $t > d_{t,\text{max}}$ then
                $v^*_{t,2}.append(A_{\text{pos}} \cdot pos)$
                valid $\leftarrow \bot$
            end if
            if valid = $\top$ & $t_1 = h_2$ then
                append $(h^*_1 \cdot A_{h,\text{recode}}, t^*_2 \cdot A_{t,\text{recode}})$ to $B_{\text{out}}$
            end if
        end for
    end if
end for
```

to be encoded are also fixed in the executable query plan and can be hardened after the physical optimizer created it. Only the recoding parameters A'_h and A'_t must be additionally passed to the operators. How this is done completely depends on the actual DMS implementation. Furthermore, the selection of the new A depends on the actual system error model, so we must leave this to future work.

The choice between either online or offline error detection and between the above three methods for hardening query processing are merely three extremes in a broader configuration space. In principal, we could also conduct error detection not in all operators, as in the Continuous Detection approach, but only in a subset of operators. This could be used to trade throughput performance against reliability. Also, in practice there are more problems to be considered. For instance, there may be historical data which is accessed only very infrequently, which is called *cold* data. In contrast, data which is accessed very frequently is called *hot*. Employing only Continuous Detection could leave cold data subject to potentially many more bit flips than hot data, as it would undergo error detection much less frequently. By that, a Δ-like operator, which does error detection, would still be needed to frequently perform such actions on cold data, where the frequency again would be determined by an actual system's error model. This could be an *offline* operation in the sense that a monitoring component in the DMS determines hot and cold data and periodically, or based on an actual error model, performs background error detection on cold data. Since actual error models are not available, we leave this aspect to future work.

5.2.4 Miscellaneous Processing Aspects

In addition to the above considerations for detecting transient bit flips in base columns and intermediate results, we now examine some side aspects regarding those opportunities. We will discuss the following side aspects in the context of query processing:

1. index structures,
2. compression, and
3. error code adaptation.

Hardened Index Structures: In Sections 4.1.5 and 4.3, we discussed (hardened) index structures, which are of course important for the processing of queries. For instance, indexes are used for index scans or in dictionaries for fast retrieval of a value's dictionary index, or fast dictionary lookup from index to value. In the former case, for scans, the index is used by the physical incarnation of the σ operator. Operations on hardened indexes are the same as on non-hardened ones and they provide in principal the same interfaces. For the latter case, dictionaries, the indexes are typically required before and after the query. Before the query, selection predicate values are replaced by the dictionary indexes. After the query, for the dictionary coded columns in the result table, the dictionary indexes in the intermediate results must be replaced by the actual values again for presenting them to the user. On these occasions, the hardened index structure operations are used. Consequently, regarding query processing, hardened index structures for bit flip detection can be used the same as non-hardened ones. Furthermore, since these are typically only *secondary* access paths, the primary access path can be used as fallback. For instance, during index scan when there is data or structural corruption inside an index so that it cannot answer the query, then a normal column scan could be used. This would accomplish Requirement $\mathcal{R}4$ in the sense that the query can be answered without interruption. Also, since e.g. the MAL interpreter (cf. Section 5.1) can choose the concrete operator implementation at run-time, anyways, it can easily pick up the appropriate alternative when a bit flip was detected.

Hardened Compression: In Sections 4.1.3 and 4.2.2, we discussed lightweight data compression and the combination with AN coding. Regarding query processing, researchers proposed to keep data compressed as long as possible and to decode only when it is absolutely necessary [4, 60]. This is much like our Continuous Detection approach, where there is only hardened data, which is never decoded. Graefe and Shapiro show that for most if not all database operators, data need not be decompressed [60]. Some researchers even propose to re-compress data on the fly [65, 99]. By that, AN coding and lightweight integer compression can allow for pervasive hardening and compression at the same time, throughout the whole query life cycle. However, not only the compression scheme may change, but also the *data domain* of the logically compressed data. By that, we may also have to adapt the AN coding parameter appropriately to match this new data domain, choosing an A e.g. from Table 3.6 or Tables A.1 to A.4. For instance, re-compression might result in a smaller logically compressed data width, which may in turn lead to a smaller A giving the same guarantees as before. Consequently, the physically compressed data width can be reduced twofold.

Hardening Adaptation: We already discussed at a few occasions, that recoding of data can be necessary. In Section 3.5.2 we showed that recoding AN hardened data boils down to a single multiplication (cf. Equation (3.63)), which makes it a simple and fast task. One important question regarding this is, when to actually further harden data, i.e. the temporal aspect. For base data, we argued about adapting to hardware aging effects. This should at best happen *before* the bit flip weights exceed the codes' guarantees, which can be e.g. the minimal detectable bit flip weight (η, cf. Equation (3.50)). Regarding query processing, recoding may happen while operators emit new intermediate results, as we just discussed for compression. The remaining question here is, *which* A to actually choose. Throughput the thesis, we argued that concrete error models are not really available, yet (cf. Section 3.2.3). We discussed how to obtain golden As (cf. Definition 10) and the process of deriving concrete error models (cf. Section 3.2.4). However, the missing bits are still the actual hardware error model and the means to derive or measure the actual bit flip weights. We will cover this aspect again in the future work Section 7.1.

5.2.5 Summary and Conclusions

In this section, we provided the first three main contributions of this chapter. We presented three opportunities for *when* to execute error detection during query processing. To the best of our knowledge, we are the first to include arbitrary bit flip detection in query processing. We introduced *Early Onetime Detection*, *Late Onetime Detection*, and *Continuous Detection*. The first two require an additional Δ operator which does error detection and decoding on a single column. While Early Onetime places Δ before any operator which fetches base data, Late Onetime places Δ as late in the query as possible. Special cases must be respected, like some arithmetic operations and multi-input operators with differently hardened data, where Δ must be placed before these operators. The third approach, Continuous Detection, does not require the additional Δ operator. On the one hand, in contrast to the first two approaches, this one requires to adapt all physical operators to include error detection primitives. On the other hand, Continuous allows to recode hardened data in each operator. Our second contribution is that we showed for the two example operators, mentioned earlier, how to adapt them for incorporating error detection primitives. Finally, we discussed three important side aspects of hardened query processing. First, hardened index structures are used as before, while we can simply exchange the non coding-aware variants with hardened index structures. In the case of detecting corruption in an index structure, index look-ups can fall back to column scans. Second, data compression is used extensively in query processing, where decompression is postponed as much as possible. Data can even be re-compressed to achieve smaller data widths. Fortunately, AN coding allows to easily adapt to the changing data widths. Third, we discussed the temporal aspect of hardening adaptation. There are no concrete hardware error models, especially concerning hardware aging effects leading to varying bit flip rates and weights. Therefore, we leave this to future work. In total, our discussions show that the query life cycle is not changed. The QEP either does not need any adaptation for Continuous Detection, or we can easily add the Δ operator for Early or Late Onetime Detection after the QEP was created by the optimizers.

5.3 HARDENED INTERMEDIATE RESULTS

We presented a methodology for hardening the physical query operators, where we implicitly assumed that we materialize all intermediate values. As it turns out, materializing intermediate data is a nontrivial task on its own, which we will discuss in the first part of this section. Especially with SIMD execution, a few alternative approaches for materialization exist. As we will see for our two evaluation systems, the tradeoff between speed and reliability must be made. In the second part, we will address the fact that contemporary in-memory DMSs also use special intermediate result formats like bitmaps, to avoid materializing data and the accompanying costs of writing that data back to main memory [1, 33, 195]. There, we will consider the performance of bitmaps when we apply our selected error coding techniques. As we will see, on the one hand it is either quite costly in terms of storage to harden bitmaps using AN coding. On the other hand, the SDC probability may be too high when hardening with XOR checksums.

5.3.1 Materialization of Hardened Intermediates

The first option is to always materialize hardened values or (V)OIDs, which we cover in more detail in the following. At certain points, data must be materialized anyways, at least when presenting results to the user. The scalar case is rather trivial, used e.g. in work from Willhalm *et al.*, who extract and store hit indexes individually, i.e. they do not use vectorization at this point [189, 190]. In contrast, for vectorized materialization, we have different options. Here, the important problem is the partial materialization of data, i.e. when not the whole vector contents should be written to memory. This is also called *selective store* and is used especially when filtering data, which already obtained some attention [143, 144, 195]. The challenge with selective store is that some data elements are discarded and to save space, the intermediate result is compacted to only contain the positively filtered data elements.

There are vector shuffling operations which compact the matching values into a consecutive list, with the remaining upper vector elements set to zero. This is shown in Figure 5.8, where the data *flow* is from middle to top (first part) and from middle to bottom (second part). The flow is also indicated as dashed line on the right half of the Figure. *Lookup table*s need to be precomputed for each data type and the selection mask from the vector comparison operation (= in this case) is used as table index to select a *shuffle mask*. This is then used to shuffle the elements from the original *data vector* and the outcome thereof is the *result vector*. The lookup tables store shuffle

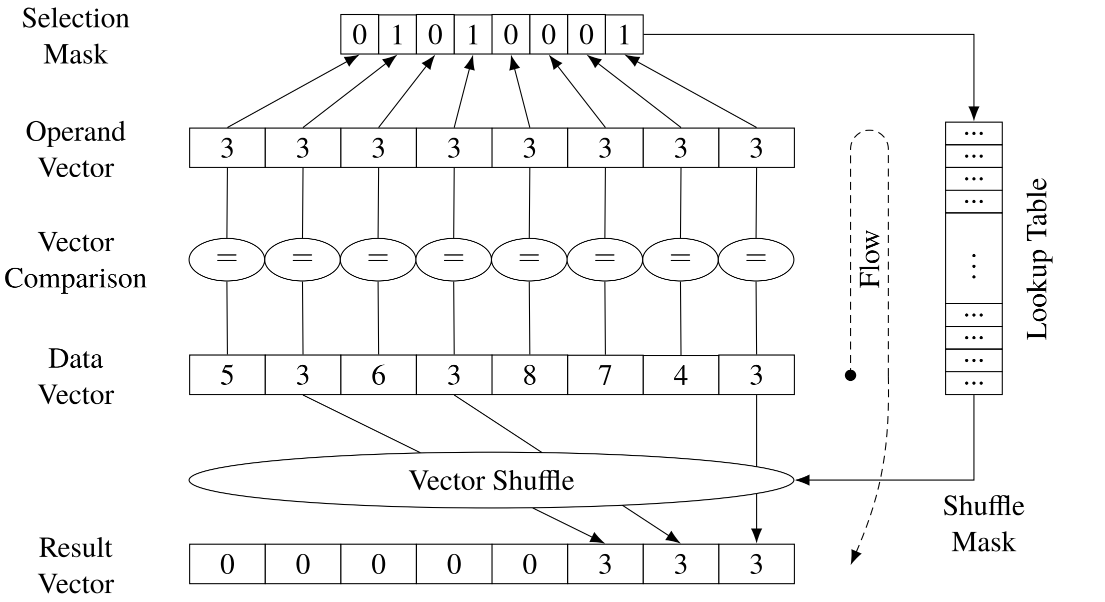


Figure 5.8: Lookup-table materialization of a vector filter result. The shuffle vector method is special in that it can set data elements to zero.

	Vector width $\lvert\mathbb{V}\rvert$					
$\lvert\mathbb{D}\rvert$	128		256		512	
	N	S	N	S	N	S
8	2^{16}	1 MiB	2^{32}	128 GiB	2^{64}	1 ZiB(a)
16	2^{8}	4 KiB	2^{16}	2 MiB	2^{32}	256 GiB
32	2^{4}	256 B	2^{8}	8 KiB	2^{16}	4 MiB
64	2^{2}	64 B	2^{4}	512 B	2^{8}	16 KiB

Table 5.1: Numbers of entries and total sizes of straightforward lookup tables for vectorized materialization. (a): 1 Zettabyte (ZiB) = 2^{70} bytes. $N \overset{\mathsf{N}}{\equiv} (T_{\mathbb{V},\mathbb{D}})$. $S \overset{\mathsf{S}}{\equiv} (T_{\mathbb{V},\mathbb{D}})$.

masks that are as wide as the vector type $\mathbb{V}$, i.e. of size $|\mathbb{V}|$ bits, for any possible combination of filtered elements. We denote $T_{\mathbb{V},\mathbb{D}}$ as the lookup table for a vector type and data element type. For each lookup table there are

$$N(T_{\mathbb{V},\mathbb{D}}) = 2^{|\mathbb{V}|/|\mathbb{D}|} \tag{5.1}$$

entries per lookup table, which results in a size of

$$\begin{aligned} S(T_{\mathbb{V},\mathbb{D}}) &= N(T_{\mathbb{V},\mathbb{D}}) \cdot \frac{|\mathbb{V}| \quad [\text{bits}]}{8 \quad [\text{bits}/\text{Byte}]} && (5.2)\\ &= 2^{\frac{|\mathbb{V}|}{|\mathbb{D}|} - 3 + \log_2 |\mathbb{V}|} \quad [Bytes]. && (5.3) \end{aligned}$$

For some instances of lookup table $T_{\mathbb{V},\mathbb{D}}$, Table 5.1 lists concrete values for currently available vector widths and data element types. Here, it becomes apparent that for some combinations this approach is infeasible. For 8-bit data, the tables would be 1 MiB, 128 GiB and 1 ZiB (Zettabyte) large, for 128-, 256-, and 512-bit vector widths, respectively. While the first is not necessarily an issue for server-grade systems, the latter two are simply way too large. The same goes for 16-bit data elements in 512-bit vector widths. Although CPU cache sizes are also increasing permanently, the combinations 8/128, 16/256, and 32/512 could be too large, as well, especially when multi-threading is taken into account and when some cache levels may be shared between multiple cores. Polychroniou *et al.* only consider AVX2 with 32-bit data elements, where the permutation lookup table size is reasonable with 2^8 elements and 8 KiB memory footprint [143]. They obtain a permutation mask to move all matching (32-bit) data members to the leading positions. Then, they use the `_mm256_maskstore_epi32` instruction, which simply skips those elements in the output vector which are not set in the given mask. As we discussed already in Section 4.1.3, assuming only 32-bit integers is not a valid option, since many more different physical data widths are used. To this regard, we see that not all SIMD ISAs support shuffling for any data width across the whole vector width. For some ISAs there are multiple lanes per vector across which the data elements cannot be easily shuffled. For instance, AVX implementations use 2 128-bit lanes, across which no byte-wise shuffling operations exist, but only inside each lane the bytes can be shuffled. Such a byte-wise shuffle across lanes must be emulated using multiple instructions. This makes it very difficult to find optimal selective store for all the data widths which a database may support and the CPU's available SIMD ISAs. Furthermore, for filtering data, the various SIMD ISAs may offer vectorized comparison operators with different return types. For instance, SSE and AVX return

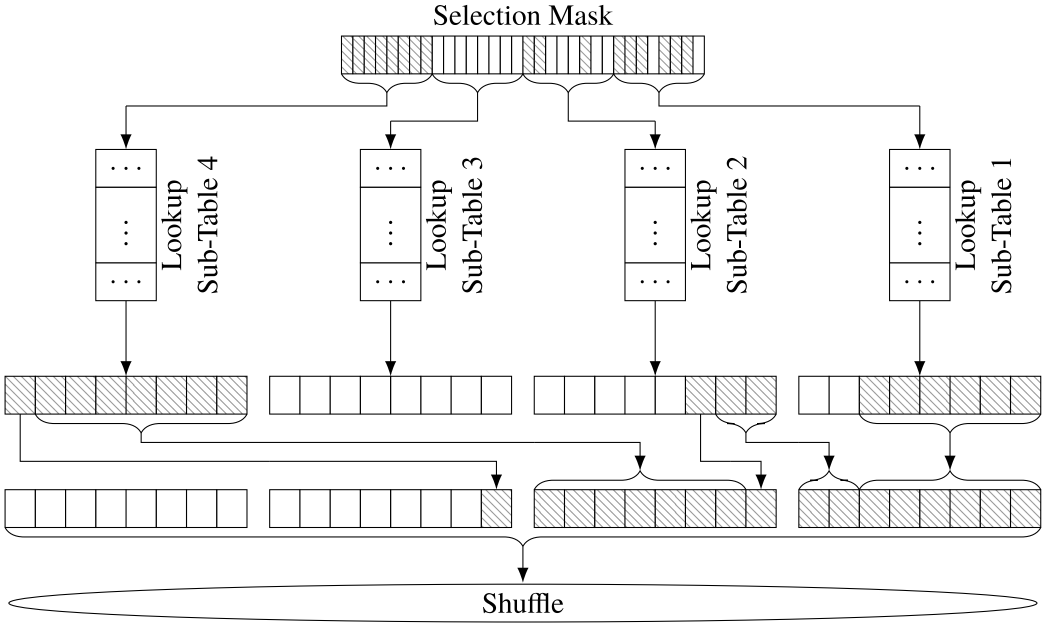


Figure 5.9: Creation of an AVX 256-bit shuffle mask from four lookup sub-tables for 8-bit data.

only whole vectors where the data units' elements are set to all-ones or all-zeros depending on the comparison. In contrast, the AVX-512 comparisons always return a mask with a single bit per data element. This results in differently complex materialization algorithms. For SSE and AVX, the vector selection masks must be translated into a lookup table index, while for AVX-512 the selection mask can be directly used as lookup table index.

We offer two solutions to the problem of lookup table method for arbitrary data widths supported by the SIMD ISAs. First, we only use a subset of the lookup tables, to reduce them to a viable size. Second, we map all shuffles to smaller vector widths. For our first solution, we split up the huge shuffle entry tables into smaller sub-tables, as depicted in Figure 5.9. The number of these tables must be a divisor of the number of values per vector, e.g. for AVX2 there are 32 8-bit values per vector and we create four sub-tables, each storing 256 shuffle sub-entries of size 64 bits. For this case, each sub-table is 2 KiB large, which results in a total of 8 KiB, in contrast to the 128 GiB for the naive approach. The sub-tables are constructed as follows. A shuffle vector contains in each position the information, which of the elements (bytes or words) from the original vector should be put into that position. A feature of the SSE and AVX shuffle instructions is, that when the MSB is set (to one), then this position is filled with zeros. Now, each sub-table contains all the possibilities for the appropriate sub-mask how the elements need to be shuffled for selective store. In our 8-bit case, there are four sub-tables, so each sub-mask contains 8 entries and there are $2^8 = 256$ possibilities for how the sub-selection mask looks like and therefore 256 entries in the shuffle lookup sub-table. This is depicted in the upper part of Figure 5.9. Then, to do the selective store, we construct the complete shuffle vector out of the sub-entries. Therefore, we fetch the first shuffle sub-mask from the first sub-table. This may contain leading all-one bytes `0xFF`, which indicates that not all values represented by this sub-mask are materialized. In Figure 5.9, such *non-materialization* entries are indicated as white rectangles, while the patterned rectangles are *materialization* entries for values to be stored. Then, we have to overlap this first shuffle entry with the next one from the second sub-table. In Figure 5.9, the first shuffle sub-mask from sub-table 1 contains 6 materialization entries, the second sub-mask contains 3 materialization entries, while the third sub-mask contains only non-materialization entries, and finally the fourth sub-mask

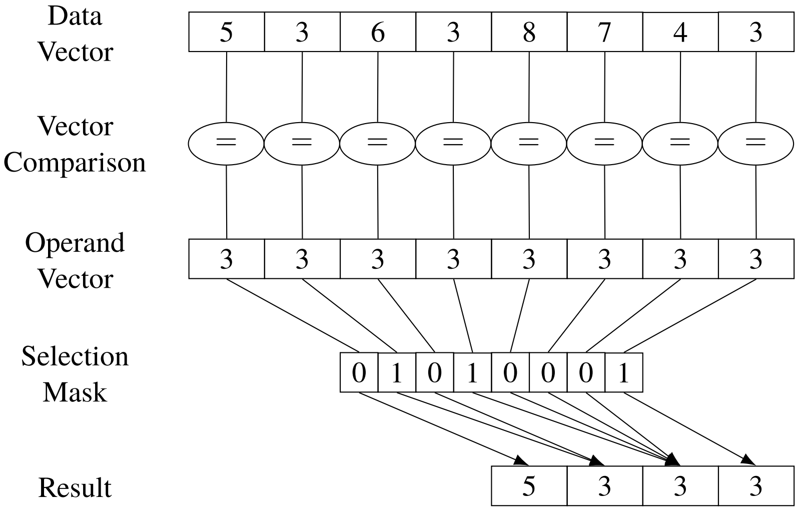


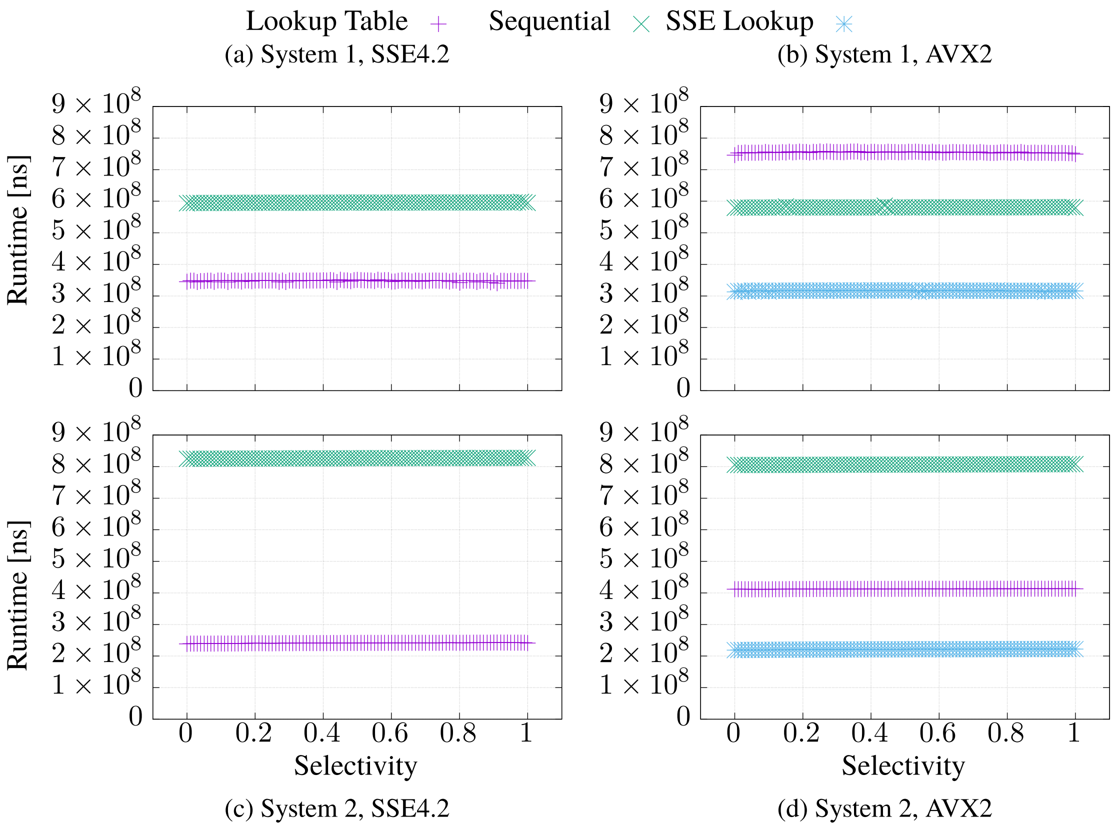
Figure 5.10: Sequential materialization of a vector filter result. All values are materialized but non-matching ones are overwritten. Since this method unconditionally writes out all values, also the last (left-most) value which does not match ($5 \neq 3$) is materialized.

contains only materialization entries. All these sub-masks must be packed together, to obtain the complete shuffle mask which is then used to shuffle the original value vector for the selective store, as indicated in the bottom part of Figure 5.9. On the one hand, this dramatically decreases the total size of the lookup tables, but, on the other hand, this requires more operations to combine the multiple shuffle sub-masks to one.

Our second solution to the problem of huge shuffle tables is to use shuffle operations from the shorter vector sizes and overlap the selective writes accordingly. For instance, for AVX and AVX-512 we would issue two and four selective store operations on SSE-sized 128-bit vectors, respectively. The output pointer which marks the start of a vector to be written is incremented only by the amount of materialized values, which requires unaligned vector store support.

Zhou and Ross propose to sequentially write out the data elements of the vector which match a given filter [195]. Figure 5.10 depicts the idea of their work (data flow is from top to bottom). To avoid branching by a conditional test for the state of the filter bit(s), they simply write out all values consecutively and increment the output data pointer to which they write by adding the filter bit from the result mask. That bit is either one or zero, whereby the output pointer is not incremented for non-matching elements, so that these are overwritten. This can further be improved by unrolling the whole loop at least for a single vector, which is easy because the actual data type and the vector size are known at compile time.

We will now compare our 2 solutions for the arbitrary shuffle masks against the sequential write out method from Zhou and Ross. For the lookup table approach, we use our partial lookup method only for the larger lookup tables. In particular this affects SSE (128-bit masks) for 8-bit data, where we use two sub-tables, and AVX (256-bit masks) for 8-bit and 16-bit data, where we use four and two sub-tables, respectively. The other data and vector width combinations use a single lookup table. Again, we measured throughput numbers for the two systems from Table 3.7 for materializing 1 Billion values, with the results shown in Figures 5.11 to 5.14. There, we measured the runtime for writing data from one array into another based on a predefined selectivity. For that, we generate a small array containing bit masks where over the whole array we set as many random bits to one to meet the target selectivity. We increase the selectivity in steps of 1%. For SSE,



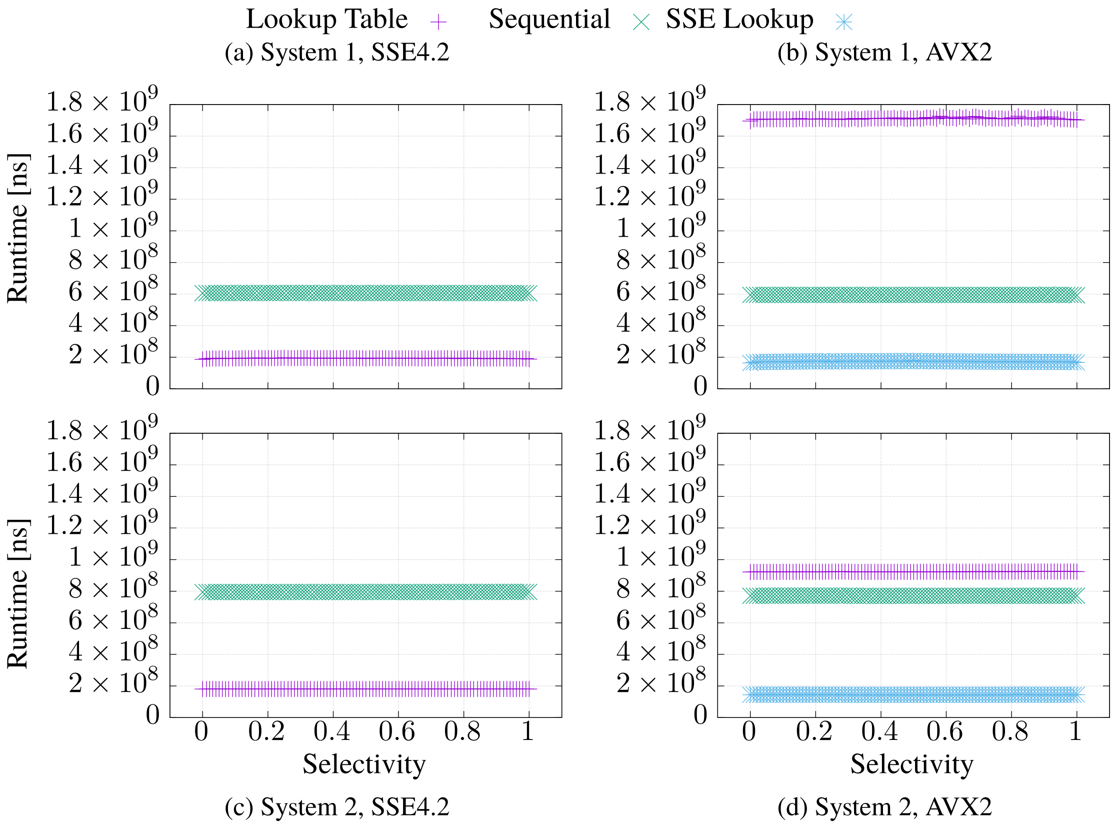
(a) System 1, SSE4.2

(b) System 1, AVX2

(c) System 2, SSE4.2

(d) System 2, AVX2

Figure 5.11: Materialization of 8-bit data



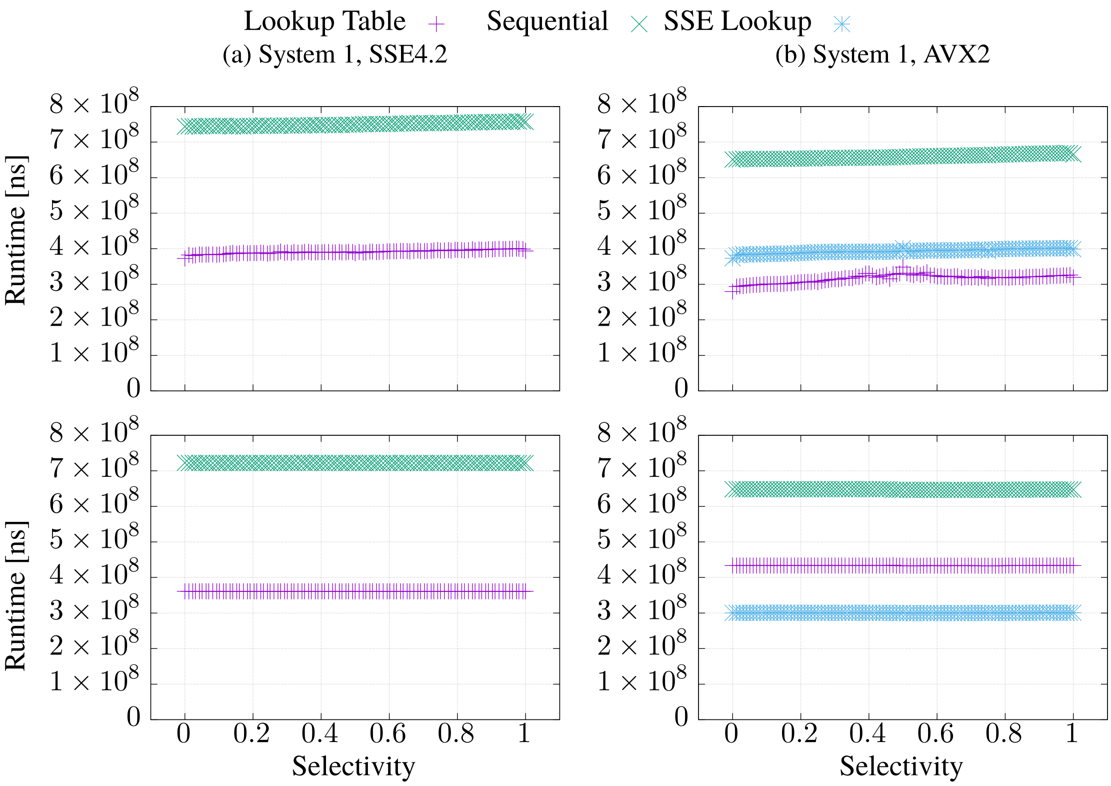
(a) System 1, SSE4.2

(b) System 1, AVX2

(c) System 2, SSE4.2

(d) System 2, AVX2

Figure 5.12: Materialization of 16-bit data.



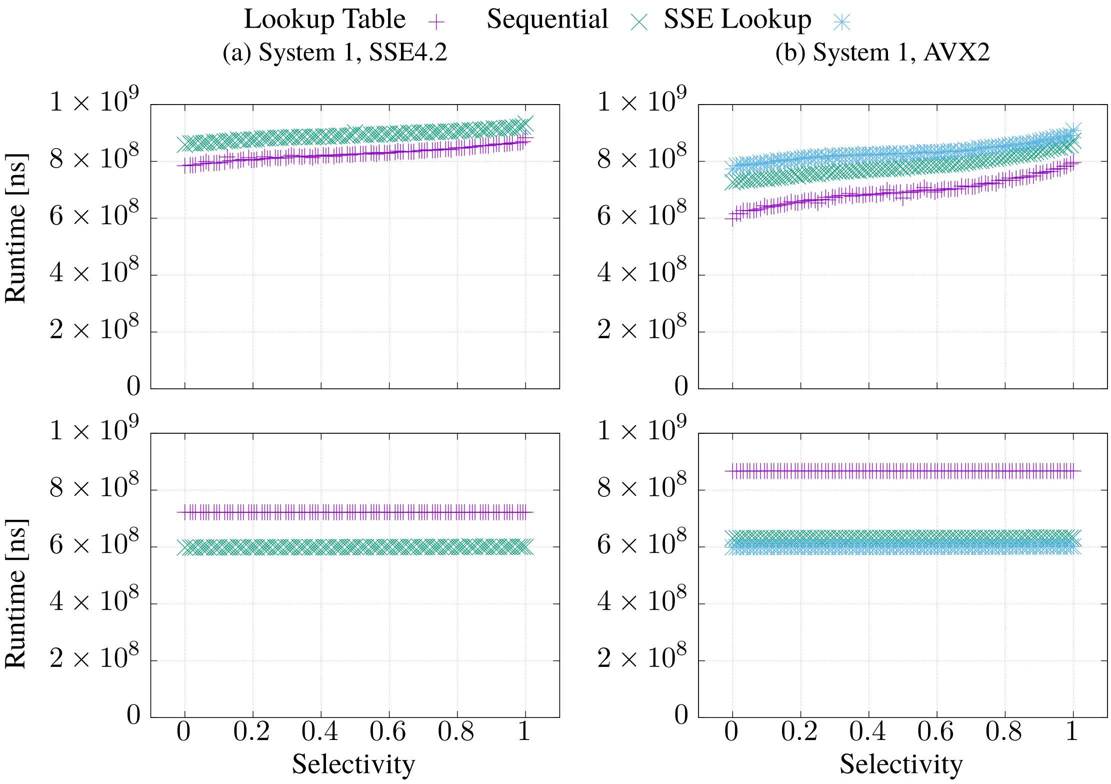
(a) System 1, SSE4.2

(b) System 1, AVX2

(c) System 2, SSE4.2

(d) System 2, AVX2

Figure 5.13: Materialization of 32-bit data.



(a) System 1, SSE4.2

(b) System 1, AVX2

(c) System 2, SSE4.2

(d) System 2, AVX2

Figure 5.14: Materialization of 64-bit data.

we only compare the lookup table approach against the sequential one from Zhou and Ross [195], while for AVX2 we additionally compare the SSE lookup approach, where we essentially only use the wider load operations. From the results we can draw the following conclusions. For SSE, the lookup table approach generally wins over the sequential approach, with the only exception being the 64-bit cases. Otherwise, the lookup method is faster than the sequential one by factors of $\approx 2 \ldots \approx 4$, even when we use two sub-tables in the 8-bit data case. For the 64-bit cases, on system 1 the lookup technique is only marginally faster, while on system 2 it is even slightly slower. For the lookup table approach, the additional operations for computing a shuffle mask out of several smaller ones introduce lots of overhead. We can see this for the AVX2 and 8-bit and 16-bit data, where the lookup table is substantially slower than the SSE variant, although AVX2 operators on double vector widths. AVX is only slightly faster than SSE for the 32-bit and 64-bit cases on system 1. On system 2, these are still slower than for SSE. Finally, using the SSE lookup for AVX2 vectors is only slightly faster than just using SSE, so this brings only very little gains, except for 64-bit data on system 1. Regarding the sequential write out method, using AVX2 does have nearly no effect, as the runtimes are typically marginally faster compared to SSE.

In total, using just SSE already delivers very high performance and for that the lookup table approach is mostly the fastest. Splitting up the lookup tables introduces high overheads for larger vector widths and when the table lookup approach is desired, the SSE lookup should just be used. However, regarding bit flips, reliability-wise the case is different: Since the lookup table approach requires many tables (for each value width) which reside in main memory, the problem of data integrity for these tables arises. For both approaches the code integrity must be maintained anyways, which is out of the scope of this thesis. The shuffle entries must be verified over and over again, otherwise simply the wrong data could be materialized, because corrupt shuffle masks might be used. This, in turn, may lead to false positives, false negatives, or missing data in the enclosing queries. Consequently, in the remainder of this thesis we will assume the iterative approach, because although it is mostly worse performance-wise, it offers better reliability from the database system perspective. We leave the problem of hardening lookup tables to future work and will give some more detail in Section 7.1.

5.3.2 Hardened Bitmaps

Bitmaps are bit vectors, where each bit corresponds to the value at the same position in the data (intermediate) column. They are typically used to represent results of filter operations in a compact format [1, 33, 195]. Then, a bit set to one means the corresponding value satisfies the filter predicate, while a bit set to zero implies the opposite. Bitmaps can be encoded with both AN coding or XOR checksums and we first discuss using AN coding. On the one hand, as we could see for memory consumption (cf. Figure 3.25), AN coding of individual bits is very space-costly. On the other hand, we can view a bitmap as a vector of numbers (limbs) and encode each limb individually, where the memory costs are lower. Frequent operations on bitmaps are the intersection (bit-wise AND) or combination (bit-wise OR) to aggregate multiple filter results on the same table [1, 33]. Executing these bit-wise operations on AN coded data does *not* result in valid code words again. Consequently, each limb needs to be decoded before the bitmap operation and the result needs to be encoded again, afterwards. In contrast, using XOR checksums is quite straight forward. Typically, the whole bitmap is read anyways, and an XOR checksum, appended e.g. after the bitmap, can be computed easily in parallel, during a bitmap operation. This can happen both for the existing bitmaps for detection and for the resulting bitmap for hardening. To investigate the impact of using either coding for bitmaps, we measured bitmap intersection on both measurement systems. We use simple arrays with randomly set bits, since the actual result

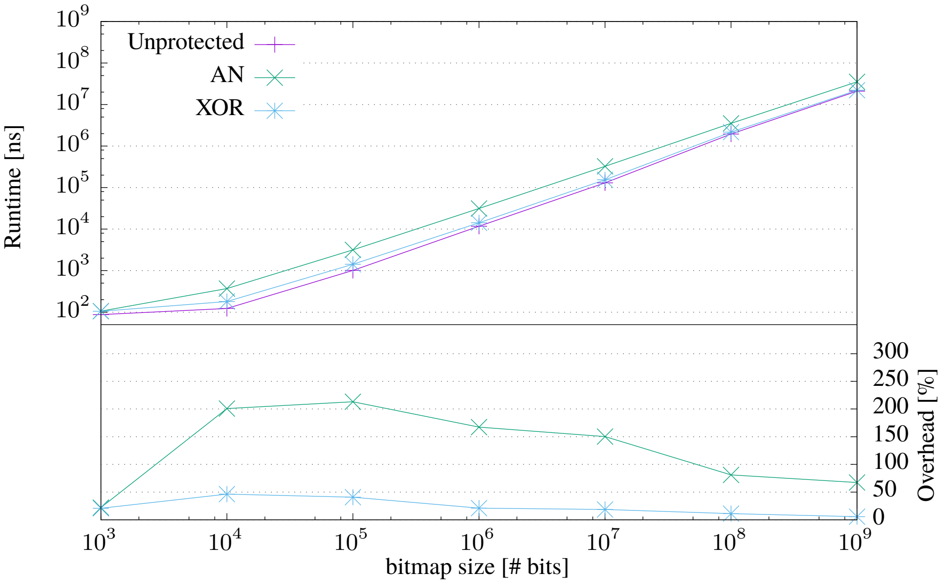


Figure 5.15: Comparison of bitmap intersection (AND-ing) runtime in [ns] (top) and overhead in [%] (bottom) between unencoded and AN hardened bit vectors on system 1.

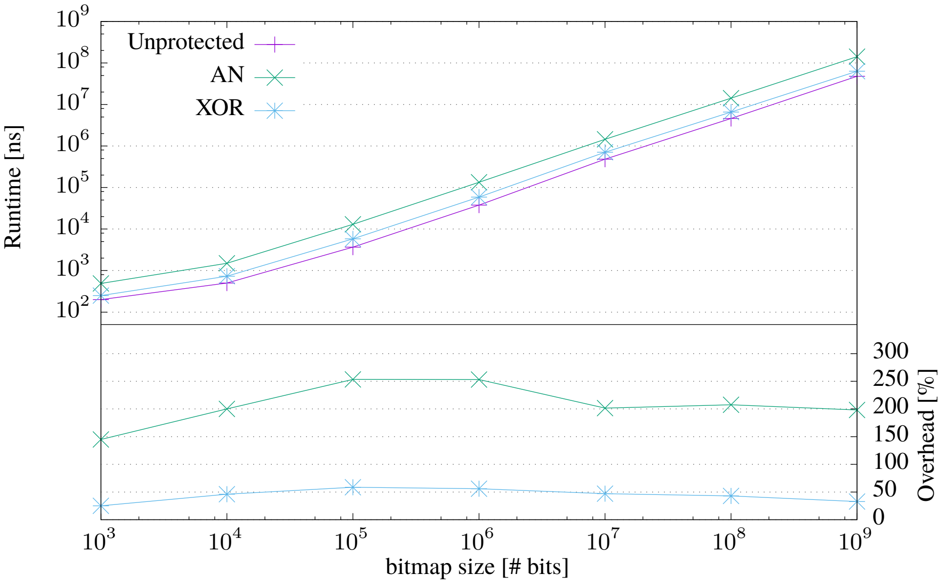


Figure 5.16: Comparison of bitmap intersection (AND-ing) runtime in [ns] (top) and overhead in [%] (bottom) between unencoded and AN hardened bit vectors on system 2.

Approach	Advantages	Disadvantages
Early	• No adaptation of physical operators	• Single point of error detection. • Rematerialization of complete base columns.
Late	• No adaptation of physical operators • Potentially less values that undergo detection than for Early Onetime.	• Single point of error detection.
Continuous	• Holistic approach: no Δ operator and no additional data materialization. • Can detect bit flips in main memory, interconnects between RAM and CPU, and CPU caches. • Recoding of each value possible, which allows *adaptability* at runtime.	• All physical query operators must be adapted.

Table 5.2: Advantages and disadvantages of the detection opportunities.

are unimportant for this measurement. We furthermore do not assume any compacted form of bitmaps. The bitmaps are represented using 64-bit integers, where for the unencoded and XOR protected bitmaps all 64 bits are effectively used, while for the AN hardened variant we only use 48 effectively and the remaining 16 bits for hardening redundancy. The actual A is irrelevant here. Figure 5.15 shows the results for system 1. The graph actually consists of two parts: the upper part shows the runtime on the left y-axis, directly comparing the runtimes of the unencoded and the AN hardened variants. The lower part shows the overhead of the hardened variants, computed as

$$\left(\frac{\text{runtime hardened}}{\text{runtime unprotected}} - 1\right) \cdot 100 \quad [\%].$$

We varied the size of the bitmap from 10^3 bits up to 10^9 bits. We see for system 1 that the AN hardened variant has much higher overheads than XOR, where for most of the measured sizes, the overhead is between 150% and $\approx 200\%$. The picture is even worse for system 2, shown in Figure 5.16. There, the AN coded variant has, except for the smallest case, an overhead between 200% and $\approx 250\%$. In contrast, the XOR hardened bitmap intersection has always an overhead of around 50%. As we see, the overheads for the AN coded variant are in total much higher than for the XOR coded variant. When it is desired to use encoded bitmaps, the tradeoff must be chosen between the flexibility and better detection capabilities of AN coding against the better runtime of XOR coding.

5.4 SUMMARY

In this chapter, we first shortly revised the most important aspects of the query life cycle. This includes parsing of a user query, optimizing the query abstract syntax tree (AST), generating and executing a QEP, and finally presenting the result to the user. There, we focused on the peculiarities of in-memory column stores and presented details of MonetDB query processing in more detail, including MAL plans and an example MAL plan graph.

Then, we introduced three opportunities for integrating hardening into query processing, which are summarized in Table 5.2. The *Early* and *Late Onetime Detection* methods introduce a new Δ operator which does error detection and decoding on a hardened column. Both methods place the operator after the optimizers generated the QEP. The Early Onetime method places the Δ operator right before any access to hardened data, so that the remaining query stays the same and all existing physical operators can stay oblivious to data hardening. The Late Onetime method exploits the fact that AN coding, being an arithmetic code, allows operations on code words which lead to valid code words again. This allows to place the Δ operator at the very end of the query, when all data is hardened with the same AN parameter. When this is not the case, we must place Δ before multi-input operators with differently hardened data, or before some other special cases like some arithmetic operators, or σ operators which evaluate predicates containing certain arithmetic operations. We argued that, Δ may incur lots of overhead since lots of data might be materialized a second time. We then presented a more holistic approach, which goes even further than the previous two methods, called *Continuous Detection*. Here, we require to adapt all physical operators to include error detection primitives. The Δ operator is not needed any longer for query processing, but when considering cold and hot data, we argued that a Δ-like operator for pure error detection might still be used to detect errors in cold data. Furthermore, intermediate results can be easily reencoded on the fly between operators. Hardened index structures can simply replace the coding-unaware counterparts, assuming that all base data is encoded anyways. We showed for two example operators how to include error detection primitives. Since we only deal with error detection in this thesis, we assume that operators for now return an error position vector, which stores the hardened positions at which errors were found. Future work can plug in at this point to run error correction primitives.

Finally, we discussed how hardening affects intermediate results and in particular selective stores, where only a subset of the input data is given to the next operator. Bitmaps can be used as a compact form to represent such filter results and to postpone materialization of intermediate data as long as possible. Our measurements showed that hardening bitmaps is very costly in terms of runtime overhead when employing AN coding, but this provides very good error detection capabilities, in turn. The memory consumption depends, again, on the limb size and the required detection capabilities. XOR checksums can be used to provide better runtimes, but have worse error detection performance and offer less adaptability than AN coding. This means that the actual choice of the used technique must be based on an actual system's error model. Next to using bitmap representations, selective stores can also fully materialize intermediate data. This is, from a *hardening* perspective, more efficient, since for the same minimal detectable bit flip weight, larger data requires less overhead in terms of additional bits (cf. Tables 3.6 and A.1 to A.4). Especially for vectorized code, there are multiple possibilities for materializing data, where the related work uses only a small subset of the possible combinations of data widths and vector widths. However, here the problem is that the lookup tables themselves typically reside in main memory, which means that they must also be hardened against bit flips.

The introduction of the Continuous Detection approach fulfills Requirement $\mathcal{R}4$ – *availability*. This is, because error detection is performed in an online fashion in each and every physical database operator, on each and every value which is read. This also holds for the hardened index structures we presented in Section 4.3, where error detection is performed during each and every tree traversal. Furthermore, Continuous Detection also fulfills Requirement $\mathcal{R}3$ – *adaptability* – since it allows to reencode outgoing data on the fly in each operator. To the best of our knowledge, we are the first to accomplish hardening query processing against arbitrary bit flips in main memory, even in an online and adaptable fashion.

6

END-TO-END EVALUATION

In this chapter we conduct an end-to-end evaluation in order to prove the feasibility of our methods for hardening the storage and processing layers of in-memory DMSs. We will first present our prototypical implementation of an in-memory column store in Section 6.1. Building such an evaluation system is challenging, because a huge variety of hardened and non-hardened physical operators must be supported, as we want to directly compare them using the same data and queries. Then, we will compare individual hardened operators against their unprotected counterparts in Section 6.2. Afterwards, we use all queries from the star schema benchmark (SSB) [131] to provide a holistic runtime comparison. The main contributions of this chapter can be summarized as follows:

1. We present a method for implementing a research prototype of an in-memory column store for comparing a huge variety of physical operators, which can be varied along several dimensions including physical data types, coding schemes, and sub-operations.
2. We conduct detailed measurements on individual physical operators as well as whole SSB queries. This shows how the improved AN coding performs in in-memory database operators and in actual queries when performing bit flip detection on each and every single value in both base columns and intermediate results. The results of the SSB evaluation for whole queries on system 1 were published in [87].

6.1 PROTOTYPE IMPLEMENTATION

To conduct the detailed operator and query evaluation, we chose a greenfield approach, by implementing a prototype of an in-memory column store, which mostly only deals with the aspects that we changed in the previous chapters. We call our prototype *AHEAD*, enabling adaptable data hardening for on-the-fly hardware error detection during database query processing [87].

6.1.1 AHEAD Architecture

We now describe the architecture of our prototype, which must support a multitude of physical database operator implementations. We need this to investigate the impact of data hardening on the individual operators, and on queries as a whole. We start with an overview of how queries are executed, depicted in Figure 6.1. Each query is a separate executable binary, where the QEP is hard-coded. First, *AHEAD* loads header files for the tables to be loaded, where information about each table's column names and types is stored. Then, it tries to load the necessary database files,

Unencoded		Encoded	
Logical	Physical	Logical	Physical
`tinyint`	(u)`int8_t`	`AN_tiny`	(u)`int16_t`
`shortint`	(u)`int16_t`	`AN_short`	(u)`int32_t`
`int`	(u)`int32_t`	`AN_int`	(u)`int64_t`
`bigint`	(u)`int64_t`	`AN_big`	(u)`int64_t`
`string`	(u)`char_t`		

Table 6.1: *AHEAD*'s mapping between logical and physical data types.

where it first checks whether proprietary `.ahead` files are present in the database path. Each column is stored in a separate file, which is named after the table and column name. When the *AHEAD* files are *not* present for all tables and columns, the necessary data files are loaded as generated by the SSB tool `dbgen`[1]. These files contain the data in pipe-delimited text format. The data is parsed and transformed into the physical data representation as denoted by the table header files. Here, we support both unencoded and hardened data. For the AN hardened integer types, we predefine for each type a golden A which is used to initially encode the data. In our prototype, we do not use special lightweight compression except using the smallest available register width needed to represent the logical data type. The mappings between logical data types and unprotected and hardened physical data types are shown in Table 6.1. There, we see that the hardened data types use the next larger register width, except for the `bigint` type, where we still use 64-bit data, since this is the largest scalar data type currently available. Therefore, the data domain is reduced in this case by the amount of bits which the A requires. Note also, that since we did not yet compute golden As for that data width, we simply use those golden As from 32-bit data. This is not a problem, because we are now only interested in the runtime behavior, and *not* the actual bit flip detection capabilities. After all data is loaded, the proprietary `.ahead` files are written to disk to speed up any next loading of the database files. When the *AHEAD* data files are already present, then the table header files are parsed and just the necessary `.ahead` files loaded for the needed columns. We assume that all columns are stored in a single, contiguous array. After all data is loaded to memory, the actual query is run in a loop to measure several runs. For each physical operator, statistics like runtime and hardware performance counters are collected and printed on the console for later evaluation.

The *AHEAD* prototype uses the BAT model known from MonetDB, because this model is easy to be implemented. In a nutshell, this mostly means that for each column the array containing the values and a column containing the respective (V)OIDs are referenced through a single data structure. Some operators, like the join, produce an output BAT where the head and tail of the inputs are intermixed, but the individual values can nevertheless be attributed to their respective (V)OIDs afterwards again. Although MonetDB currently implements a headless mode, where actually the BAT model is abandoned and each input and output is directly seen as column, we still stick to the BAT model. This is only a cosmetic change and does not change the expressiveness of the system. However, we chose to materialize all data, anyways (cf. Section 5.3), and by that we always create pairs of columns, containing both OIDs and the actual payloads. This is to identify any value at any point in time during the query. We distinguish between two types of BATs, `ColumnBATS`, which contain column base data, and `TempBATS`, which are intermediate results produced by the operators.

6.1.2 Diversity of Physical Operators

A big problem is that our prototype must support a huge amount of operator variants. This stems from the following facts. First, we want to compare unprotected operators without error detection primitives against hardened operators which do include them. Second, each operator must be tailored to the individual physical data types. Third, we implement both scalar and vectorized SSE variants of some operators. Table 6.2 lists a subset of the operators which we implemented in our prototype to support the whole range of SSB queries [131]. Our prototype in total supports eleven data types: four integer types (`tinyint`, `shortint`, `int`, and `bigint`), OID and VOID, strings, as well as the four AN coded integer counterparts and AN encoded OID. The type system of our prototype is designed such that it can be extended by more types. For the AN coded data types, we define a fixed set of possible parameters – the golden As from Tables A.1 to A.4. This number of

[1] See e.g. `https://github.com/lemire/StarSchemaBenchmark`.

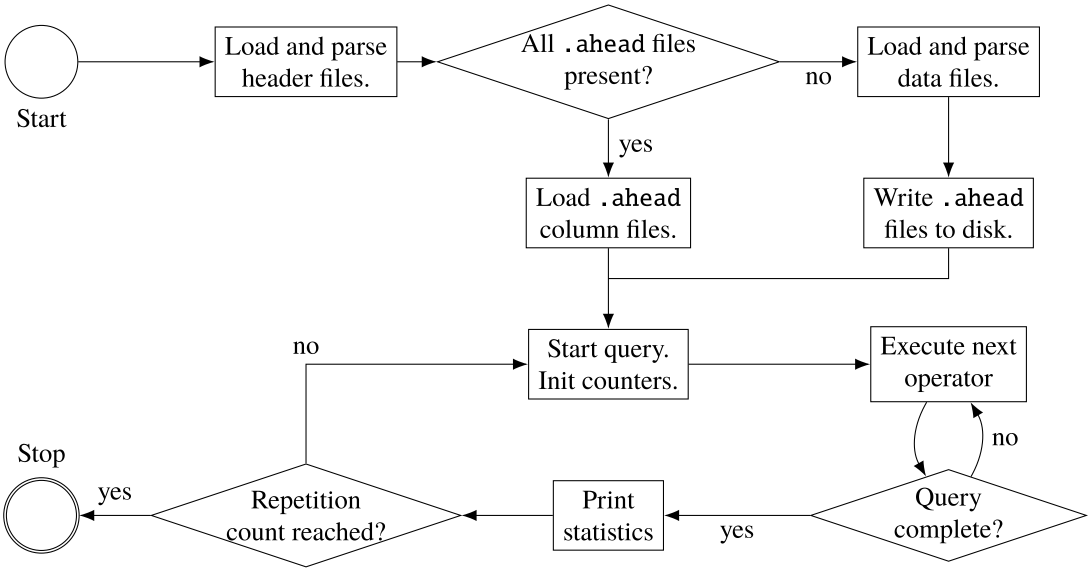

Start
Load and parse header files.
All .ahead files present?
no
Load and parse data files.
yes
Load .ahead column files.
Write .ahead files to disk.
no
Start query. Init counters.
Execute next operator
no
Stop
yes
Repetition count reached?
Print statistics
yes
Query complete?

Figure 6.1: Query execution in the *AHEAD* prototype.

Variant	Signature	# Input BATs	# Type variants	# Predicate variants
Scalar only	copy (BAT<H, T>)	1	11^2	
	matchjoin (BAT<H_1, J>, BAT<J, T_2>)	2	11^3	
	fetchjoin (BAT<void, oid>, BAT<void, T_2>)	2	11^1	
	hashjoin (BAT<H_1, J>, BAT<J, T_2>)	2	11^3	
	groupby (BAT<H, T>)	1	11^2	
	groupby (BAT<H, T>, BAT<void, oid>)	2	11^2	
	sum_grouped (BAT<H, T>, BAT<void, oid>)	2	11^2	
	Theoretical upper bound:		3157	
Scalar / SSE	check_and_decode_AN (BAT<H, T>)†	1	11^2	
	select (BAT<void, T>, pred)	1	11^1	6
	select (BAT<void, T>, pred1, pred2)	1	11^1	36
	arithmetic (BAT<H_1, T>, BAT<H_2, T>)	2	11^3	4
	sum (BAT<H, T>)	1	11^2	
	mul_sum (BAT<H_1, T>, BAT<H_2, T>)	2	11^3	
	Theoretical upper bound:		$2 \cdot 7359$	
Sum (Unprotected + AN hardened + Recoding)†: $3 \cdot (3157 + 2 \cdot 7238) + 2 \cdot 11^2 = 53\,141^\dagger$				

Table 6.2: Theoretical number of implementations per physical operator for the supported number of data types. †: Operator check_and_decode is counted only once with all type variants in the final sum, since it has neither an unprotected nor a recoding counterpart.

supported data types is reflected in the “# type variants” column, where the number is multiplied depending on the possibilities of the input BATs’ types. As Table 6.2 depicts, all operators take one or two BATs with a diverse number of type variants. Some operators have restrictions like the join operators, where some data types are fixed and the join types must be the same, indicated by type name `J`. The other type names are either given as `VOID` or `OID`, or as placeholders, where `H` and `T` denote the type for the head and tail, respectively. Subscripts also indicate the number of the BAT to which the type corresponds. The `detect_and_decode` operator realizes the Δ operator which we introduced for the Early and Late Onetime Detection approaches in Section 5.2.1 and Section 5.2.2, respectively. Our selection operator `select` has two variants, for a single predicate and for two predicates. The former supports any of the comparisons $<, \leq, =, \neq, \geq, >$, while the latter supports any combination of these six comparison, whereas in a real system some combinations might not make sense and be left out. The `arithmetic` operator, in turn, supports the four basic operations $+, -, \cdot, \div$. Furthermore, the upper half of the listed operators is only implemented as a scalar variant, while the lower half is implemented as both scalar and SSE variants. Table 6.2 also lists the total numbers of possible operator variants, which is in total $\approx 53\,141$. This is the total sum of the theoretical upper bounds for our set of operators, meaning that this would have been the total amount of physical operators if we had wanted to allow any type in any placeholder. This number includes the full set of scalar and vectorized operators for unprotected, AN hardened and AN recoding variants. Since the Δ operator `check_and_decode_AN` is only used for the AN hardened operators (not for recoding), it is only included once in the total sum.

To illustrate how we deal with this excessive amount of operator implementations, consider Figure 6.2, where there is a schematic figure of the selection operator. There, we visualized the compile-time configuration possibilities, which allow to generate many operator instantiations through a few operator templates. The selection operator is depicted as the gray box with some template places (triangle and semi-circle recesses) where the actual types (top) and comparison operations (bottom) can be set. From each type, we can in general choose any two to fit the head and tail type. For the head and tail types, the full range of data types is supported, and we will see shortly how we support such different types like integers and strings. As can be seen on the very left and right sides, the operator takes one input BAT and generates one output BAT. Furthermore, three variants exist for unprotected, hardening-aware, and hardening-aware-and-recoding selection, indicated as the switch in the right half of the operator rectangle. By that, we can also run unprotected operators on hardened data, as is required for the late onetime detection scheme (cf Section 5.2.2). The two hardened variants, in turn, are realized using the *same* code template and a template Boolean parameter which actually switches between the two behaviors. Furthermore, the encoded variant requires the head’s and tail’s A as input parameters, while the recoding variant additionally requires the respective Anews, shown on the left. Both are displayed dashed since they are optional and only used for these cases. Figure 6.2 shows two comparisons to be specified at the bottom, where the second is also optional. The actual use depends on whether one or two predicates are desired.

As we have just seen, supporting a certain range of data types and physical operators which come in many variants leads to a huge amount of implementations. To cope with this excessive amount, we use several techniques. First, we make extensive use of the C++ template mechanism. Each operator is implemented only once and then *instantiated* for all desired data type and potentially operation combinations. In the sources, this is done via explicit template instantiation[2]. This offloads lots of work to the compiler, which then also can apply optimizations, e.g. for the specific data types. We exploit the fact that modern compilers do quite intelligent inlining and optimize much of the template code away. Second, instead of supporting any possibly used data width,

[2] See e.g. `http://en.cppreference.com/w/cpp/language/template_specialization`.

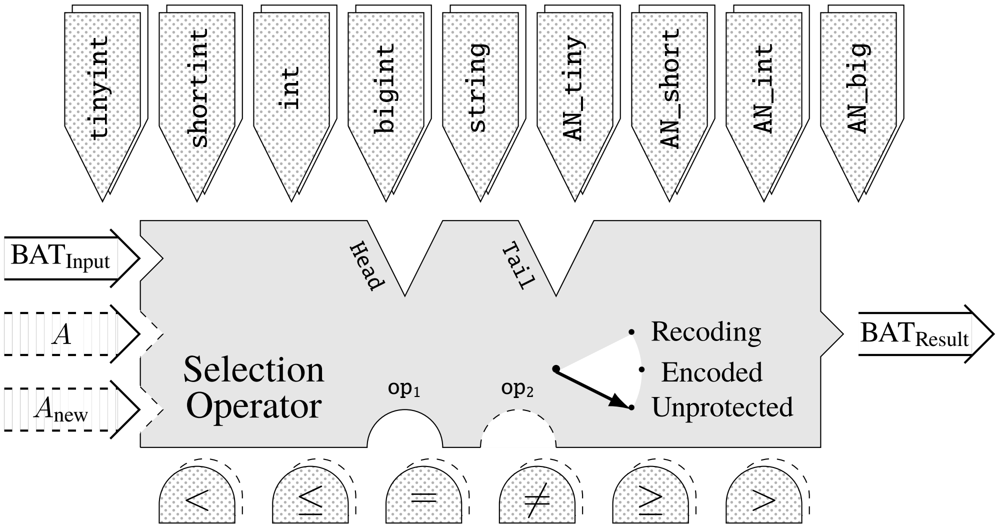


Figure 6.2: Schema of the compile-time configurability of the selection operator.

in our current implementation we use only the native implementation data types provided by C++. This implies that we implicitly do byte- and word-aligned integer compression. For the columns we then statically assign the smallest integer type which is needed to represent all values. In order to support the various hardened data types, we use a secondary type system, based on `struct`s. This is needed, because native types and all their aliases (`typedef`s and the like) are during template type resolution reduced to their base type. To circumvent this, we use an additional `struct` for each supported data type which is then used to instantiate all the operators for compilation. Third and finally, we implemented a SIMD abstraction layer, which in principal allows to define operators through a skeleton of template code which then calls all the respective operations for the desired SIMD ISA, e.g. SSE. This abstraction layer was also used for our microbenchmarks in Section 3.4.2. Regarding the total amount of instantiated operators, we also apply several restrictions. For instance, we assume that the selection operators are only fed with base columns, where the head type is VOID. Using the template specialization technique, we also do not allocate any memory storage for the head or tail when it is of type VOID. The BAT structure is defined such that it contains a special container member for all data types except for the VOID type.

6.1.3 One Concrete Operator Realization

To provide one concrete example, consider Listing 6.1, where we depict the unprotected variant of the `select` operator for a single predicate. The actual function is embedded in a `struct`, to support *partial* specialization of a subset of the template parameters, which we will describe below. On Line 1, we see that it takes several template type parameters. First, there is a `template<typename> class Op`, which is an abstraction from the actual operation, where our implementation supports the six comparisons $<, \leq, =, \neq, \geq, >$. Template parameter `Tail` is the input BAT's tail type. On Line 3, the `struct` is partially specialized on the input BAT's head type to be of type is VOID (`v2_void_t` on Line 2). This shows the implementation way of basically restricting the selection on base columns and this can be easily extended by either avoiding the partial specialization at this point, or by adding partial specializations for more head types. Lines 4–7 introduce type aliases (`typedef`) to shorten the code thereafter. For better readability, the type names are indented. Our type system defines for each data type a respective *selection* data type, which is used whenever an operator may return only a subset of the original values. This is required in this case, because we materialize

```
template<template<typename > class Op, typename Tail>
struct Selection1<Op, v2_void_t, Tail> {

    typedef typename Tail::type_t               tail_t;
    typedef typename v2_void_t::v2_select_t     head_select_t;
    typedef typename Tail::v2_select_t          tail_select_t;
    typedef BAT<head_select_t, tail_select_t>   bat_t;

    static bat_t*
    filter(
            BAT<v2_void_t, Tail>* arg,
            tail_t th) {
        auto result = skeleton<head_select_t, tail_select_t>(arg);
        result->reserve_head(arg->size());
        auto iter = arg->begin();
        Op<typename Tail::v2_compare_t::type_t> op;
        for (; iter->hasNext(); ++*iter) {
            auto t = iter->tail();
            if (op(t, th)) {
                result->append_head(iter->head());
            }
        }
        delete iter;
        return result;
    }
};
```

Listing 6.1: The abstract unprotected selection operator implementation

```
template<template<typename > class Op>
struct Selection1<Op, v2_void_t, v2_str_t> {

    typedef typename v2_void_t::v2_select_t v2_head_select_t;
    typedef typename v2_str_t::v2_select_t v2_tail_select_t;
    typedef BAT<v2_head_select_t, v2_tail_select_t> result_t;

    static result_t* filter(
            BAT<v2_void_t, v2_str_t>* arg,
            str_t threshold) {
        auto result = skeleton<v2_head_select_t, v2_tail_select_t>(arg);
        result->reserve_head(arg->size());
        auto iter = arg->begin();
        Op<int> op;
        for (; iter->hasNext(); ++*iter) {
            auto t = iter->tail();
            if (op(strcmp(t, threshold), 0)) {
                result->append_head(iter->head());
            }
        }
        delete iter;
        return result;
    }
};
```

Listing 6.2: String specialization of the unprotected selection operator

all intermediate results and for the head type we need to convert the data type from VOID to OID. Then follows the actual operator implementation on Lines 9–25. The operator returns a `bat_t`, (Line 9) which was previously defined to be a BAT containing the respective selection types of the input BAT (Lines 5–7). The operator's arguments are the input BAT (Line 11) and the threshold (Line 12), which must be of the BAT's tail type. Then, first the result BAT is allocated on Line 13, and space is reserved (Line 14), whereas for now we allocate enough space to accommodate all values. In practice, selectivity estimations from the query optimizers would be used to allocate as little memory as necessary. Afterwards, the operator iterates over the bat using a special BAT *iterators* (Line 15). In the sources, an operator object is created on Line 16, but in reality, since this only contains a simple comparison evaluation function, this object is never really instantiated. The iteration loop is started on Line 17, then the value is retrieved from the iterator on Line 18 and the predicate evaluated on Line 19. If it matches, only the *head* value is stored in the result BAT instead of both head and tail. We employ this small optimization, because in queries, especially those from SSB, typically not all attributes for which a selection is done are also needed for the end result. By that, we need the respective OIDs of the selected attributes only, anyways. At the end of the operator, we need to deallocate the heap-allocated iterator (Line 23) and, finally, return the result (Line 24). As can be seen from the code snippets, we did *not* employ any loop unrolling for the *AHEAD* operators.

Listing 6.2 shows the selection operator for strings, where one more template type is specialized, namely the tail type. By that, the compiler will automatically use this implementation whenever it encounters the selection operator and an input BAT with string as tail type. In principle, the operator is the same as before, except that the template definition on Line 1 lacks the tail type, that the `struct` definition defines the string type, and that the predicate evaluation on Line 17 calls the `strcmp` function, which returns an integer denoting the first difference between the two given strings, or zero when both are equal. This can be elegantly used in combination with the template comparison operator, which now compares the output from `strcmp` against zero, which is syntactically exactly the desired comparison.

The hardened operator variants are implemented as separate operators, derived from the unprotected operators by adding error detection primitives and error position vectors as indicated in Section 5.2. Using these techniques, we can instantiate all useful combinations of operators from Table 6.2 with just *49* operator code templates. In our prototype, we only employ AN hardening for integer data types, but the way we designed *AHEAD*, it can be easily extended by more error coding techniques.

6.1.4 Summary and Conclusions

In this section, we presented details of our prototypical *AHEAD* in-memory column store. We used a greenfield approach so that we could concentrate on those parts of the physical layer which we had to adapt for data and operator hardening. It supports several integer types, both unprotected and AN hardened, as well as unprotected strings. To evaluate integration of hardening we implemented all of the SSB queries and collect detailed statistics for each operator. W.l.o.g, we use the BAT model known from MonetDB as one possible main memory-centric column store implementation. One big challenge is the great diversity of operators, as we must support both unprotected as well as hardened operators, while for several of them we offered both scalar and vectorized implementations. To support all operators for SSB including all data type variants etc., we would have had to implement more than 50 000 physical operators. To handle this huge diversity, we showed how to elegantly employ modern C++ template language features, by which the number of actual templates, including some specializations, is reduced to just 49. We illustrated this abstractly for the selection operator and discussed the actual generic implementation, as well as the `string` specialization. The error detection primitives are added just as depicted in Section 5.2. In total, this shows how to handle a huge variety of physical data types, coding schemes, and operations.

6.2 PERFORMANCE OF INDIVIDUAL OPERATORS

We will now investigate the influence of including the AN error detection primitives in several operators. Therefore, we take measurements for individual operators from some of the SSB queries. Each operator is executed $10\times$ for each of the SSB scale factors $1 \ldots 10$, i.e. $100\times$ in total. For reference, Table 6.3 lists the number of tuples per table and scale factor. For table "Customer", the tuple count increases by a factor $30\,000$ per scale factor, while for table "Date" the number of tuples stays the same with 2556 entries. It contains for a period of seven years one tuple per day, including one leap year. The size of table "Lineorder" is approximately $6\,000\,000$ $\times$ the scale factor. Table "Part" scales with a ratio of about $200\,000 \cdot (1 + \lfloor \log_2 \text{scale factor} \rfloor)$, while table "Supplier" scales linearly with a factor of 2000 per scale factor. The SSB consists of a small star schema database layout, where table "Lineorder" is the fact table and the other tables are the dimension tables. In fact, it is a simplified variant of the TPC-H[3] benchmark. For all operators, we will show six graphs in total, three per measurement system (Table 3.7), showing four different metrics which are arranged below each other, and for each metric the graphs for the two systems are next to each other. The four metrics are from top to bottom: (1) the absolute operator runtimes in milliseconds in the first graphs, (2) the runtime overheads of AN coding in percent in the in the middle graphs, (3) the instruction overhead in terms of additionally retired instructions, and (4) the average instructions per cycle (IPC) per operator lower graphs. We chose these metrics to show several aspects. The runtime graphs show whether the operators scale linearly or not, which is an important property with increasing amounts of data. In addition, the runtime overhead graph shows the actual AN coding overhead per operator, which is not really visible from the absolute runtimes. This also shows how error detection scales with increasing column size lengths. The instruction overhead provides the amount of additional work in terms of additionally retired instructions. These are instructions which were completely executed and emitted from the execution pipeline and represent the actual amount of work done for the real control flow and does not count instructions done by speculative execution. The IPC is used as an indicator how the CPU schedules the additional instructions which are added by the error detection primitives. Also, we measure both normal AN error detection, as depicted in the continuous error detection opportunity from the previous chapter, as well as AN error detection including *recoding* of all materialized values. We denote the systems by their CPUs' code names Skylake (system 1) and Xeon Phi (system 2). As we will see, the simpler core architecture of the Xeon Phi and the lower core frequencies lead to quite different measurements than for the Skylake system. In the following, we will present the measurements first for selection operators with one and two predicates, then those of three different join implementations, and finally those of grouping and aggregation operators. The Δ operator does not have an unencoded counterpart and is in fact only used by Early and Late Detection. Therefore, we will in the end show the benefits of vectorization for the Δ operator.

[3] See `http://www.tpc.org/tpch/`.

Scale	Table Cardinalities				
Factor	Customer	Date	Lineorder	Part	Supplier
1	30 000	2556	6 001 215	200 000	2000
2	60 000	2556	11 997 996	400 000	4000
3	90 000	2556	17 996 609	400 000	6000
4	120 000	2556	23 996 604	600 000	8000
5	150 000	2556	29 999 795	600 000	10 000
6	180 000	2556	36 000 148	600 000	12 000
7	210 000	2556	41 995 307	600 000	14 000
8	240 000	2556	47 989 007	800 000	16 000
9	270 000	2556	53 986 608	800 000	18 000
10	300 000	2556	59 986 052	800 000	20 000

Table 6.3: Tested scale factors and table cardinalities for the Star Schema Benchmark.

6.2.1 Selection on One Predicate

We start with our comparison of the selection of one column using one predicate. All SSB queries include such selections and we take measurements from query 1.1. For the selection operator we implemented both scalar and SSE variants. We will first discuss the measurements for the scalar variant and then those for the SSE variant. It should be noted here that the "Quantity" column is stored using the 8-bit data type `tinyint` for the unencoded case and as 16-bit `AN_tiny` for the hardened case.

Scalar Execution

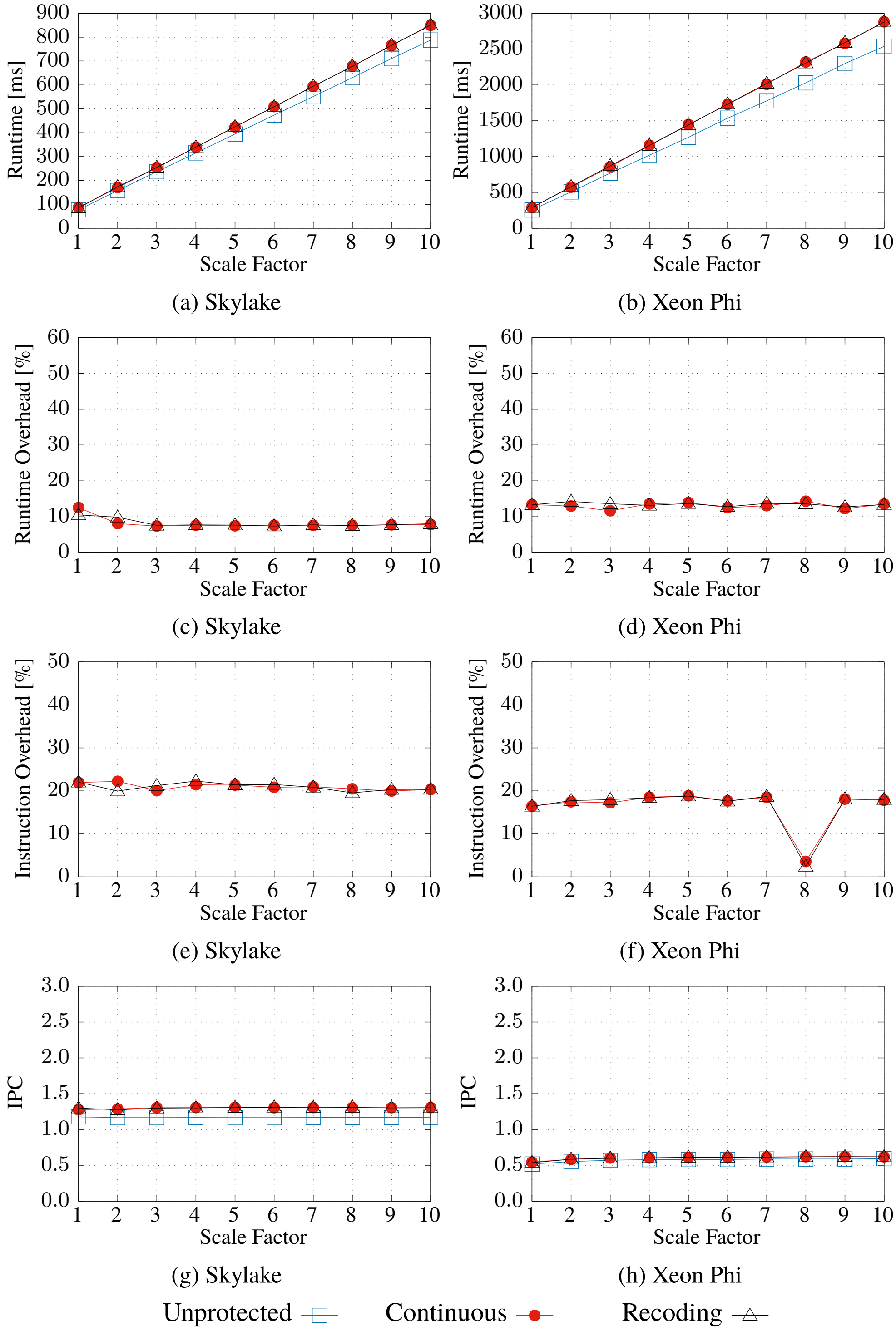
The absolute runtimes for scalar execution are shown in Figures 6.3a and 6.3b, which show a linear increase in runtime for both Skylake and Xeon Phi systems and for all three operator variants. The runtimes already indicate that the unprotected operator is slightly faster than the hardened ones. The values differ dramatically between Skylake and Xeon Phi, where the former is more than 3× faster than the latter. However, the runtime overhead, shown in Figures 6.3c and 6.3d reveals that both systems have nearly the same relative overhead of around 8.2% for Skylake and 13.1% for Xeon Phi on average, which is a difference of only 5%. The graphs also show that with varying scale factor, the overhead is quite stable. The instruction overhead, presented in Figures 6.3e and 6.3f, shows 21% for Skylake and 16.4% for the Phi on average additional work for the hardened variants, compared to the unprotected ones. Especially for Skylake, even though we have 21% and more instructions, we only have a runtime overhead of 8.2%. This means that the CPU can efficiently schedule the additional error detection instructions. Figures 6.3g and 6.3h shows that the hardened variants have a slightly higher IPC than the unprotected operator. This supports our previous statement, meaning that with a higher IPC, more instructions are executed in parallel. This in turn supports our assumptions from Section 3.5.4, that the improved AN coding using the inverse leads to multiple, independent execution paths (cf. Figure 3.36). Albeit the IPC is not substantially higher, it suffices to compensate for the additional instructions. Of course, the additional instructions have varying numbers of cycles until they finish and this must be accounted for, as well. Furthermore, while we see that the runtime and instruction overheads are quite similar for the Skylake and Xeon Phi systems, the absolute runtimes and the IPC differ quite much. The Skylake system has an IPC of 1.17 for the unprotected operator and of 1.30 for the hardened and recoding variants, on average. In contrast, the Xeon Phi system only shows IPCs of 0.57 and 0.60 on average for the unprotected and the hardened operator variants, respectively.

Vectorized Execution

Now, we consider the vectorized execution, for which we implemented SSE variants of the respective operator. The measurements are shown in Figure 6.4. Here, we can first see that the vectorization for the column filter scan is much faster than the scalar variant from Figures 6.3a and 6.3b. For Skylake, the unprotected vectorized scan on one predicate is more than $12\times$ faster than the scalar one. The hardened operators also speed up a lot, but these are only slightly more than $8\times$ faster than their scalar counterparts. This makes sense when we consider the data types again. The unprotected operator works on 8-bit `tinyint` data, where it can process 16 values per vector instruction using on SSE. In contrast, the hardened operator uses the 16-bit wide corresponding `An tiny` data type and, consequently, each vector operation can only work on half as many values as for `tinyint`. This in turn leads to a higher difference between the vectorized operators than for the scalar ones. Now, on the Skylake system, the runtimes between unprotected and hardened operator differ by as much as $51.9\ldots54.5\%$, with a deviation for scale factor 1, where the overheads are around 114%, as shown in Figure 6.4c. On the Xeon Phi system, the overhead is smaller with 35% on average for both hardened variants. However, the total runtime is again much higher than that from the Skylake system. The amount of retired instructions is much higher on the Skylake system than on the Xeon Phi system, which may also contribute to the higher runtime overhead, on the one hand. On the other hand, the IPC value is much higher for Skylake compared to Xeon Phi, so that again many more instructions are executed per CPU cycle. It is about $5\times$ as high, with values of 1.95 and 0.39 on average for Skylake and Xeon Phi, respectively.

6.2.2 Selection on Two Predicates

Next, we present the measurements for a filter column scan using two predicates. This introduces more work to do per value and we are interested in how far this affects the runtime overhead of the hardened operators. The scalar execution variants shown in Figure 6.5 are in essence the same as for the single-predicate variant. One difference is, that the runtimes are slightly lower than before. The more interesting part is the vectorized one, shown in Figure 6.6. Again, the runtimes are slightly lower than for the single-predicate variant, but now for the Skylake system, the runtime overhead decreases down to 44% on average. The IPC increases on average, with 2.6 instead of 2.3 for the unprotected operator, and with 2.4 instead of 2 for both AN hardened ones, for Skylake. In contrast, the averages for the Xeon Phi stay virtually the same with an increase of 0.01 for the unprotected and 0.03 for both hardened variants. To summarize, the more powerful Skylake system can mitigate the coding overhead when more work is done per value.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

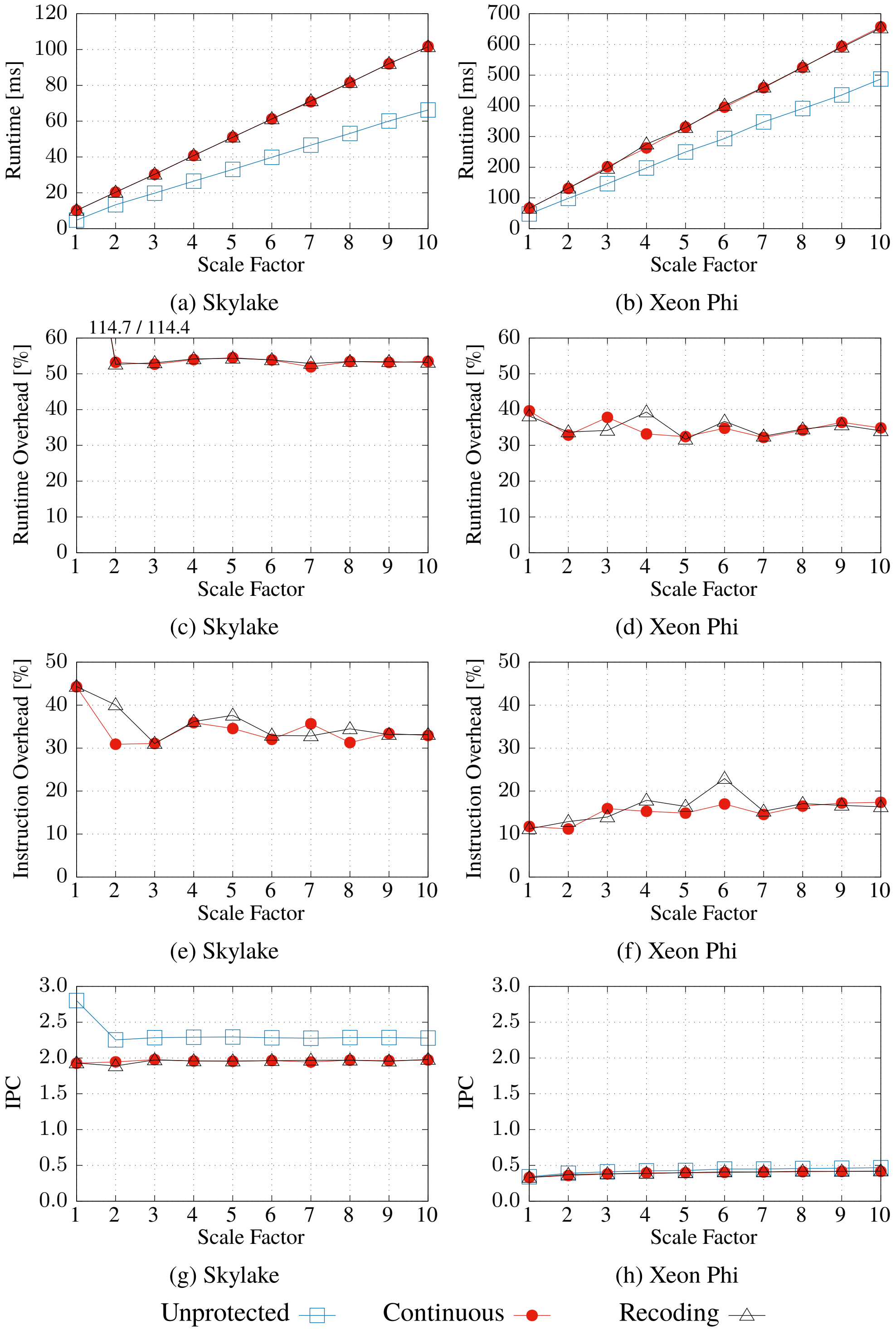
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.3: Measurements for **scalar selection** operator on table "Lineorder" with **single predicate** `quantity < 25`.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

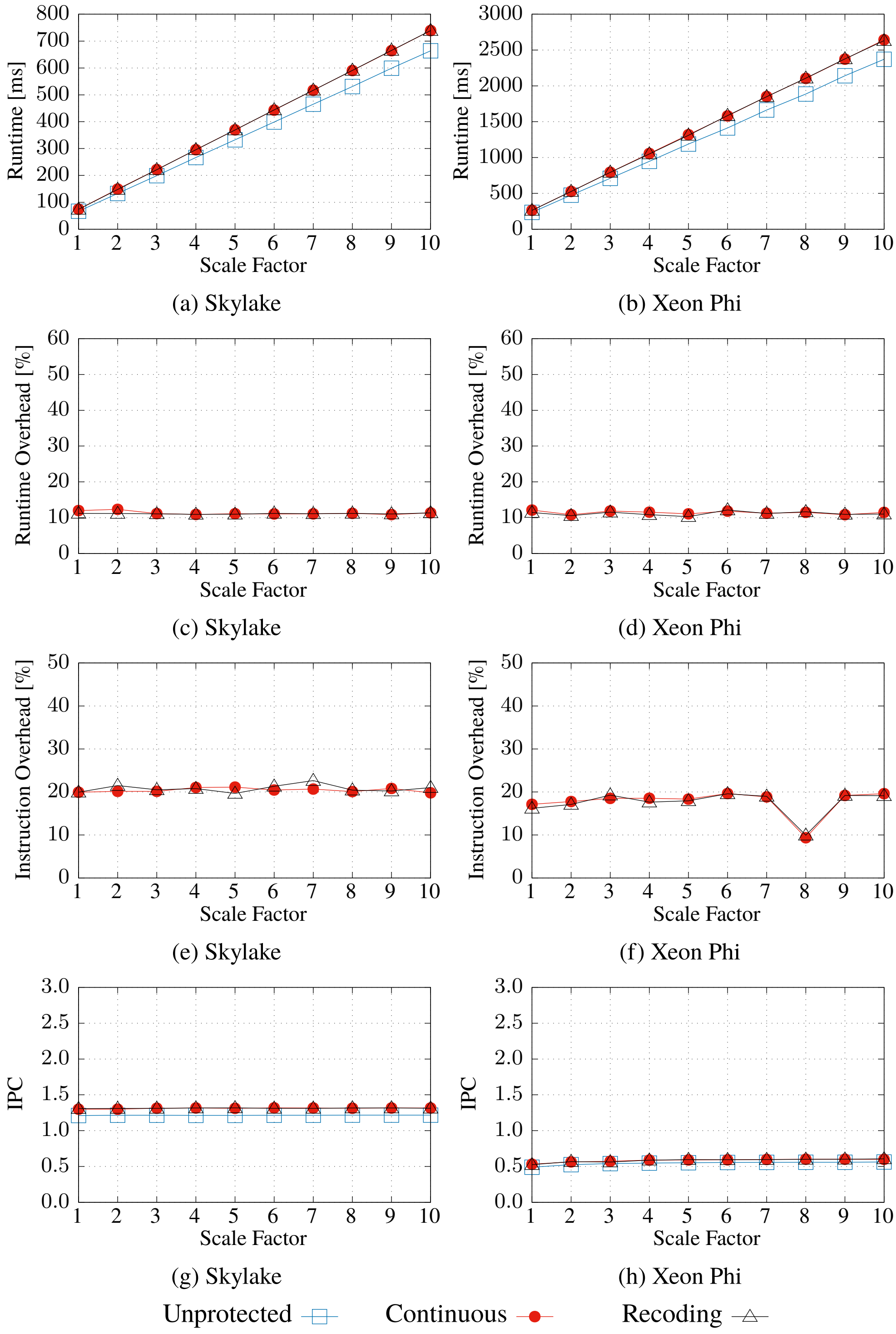
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.4: Measurements for **vectorized selection** operator on table “Lineorder” with **single predicate** `quantity < 25`.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

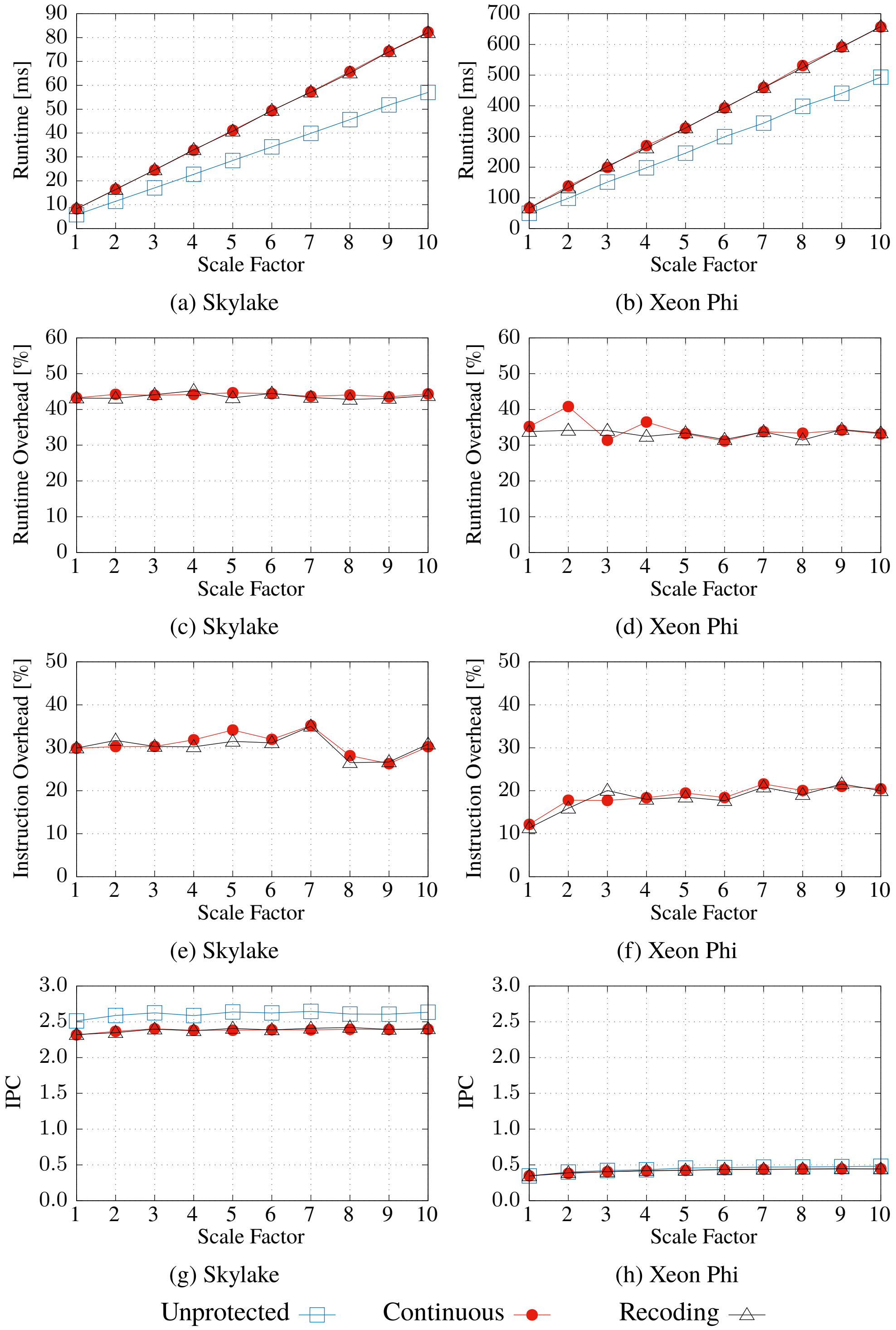
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.5: Measurements for **scalar selection** operator on table “Lineorder” with **two predicates** `1≤discount≤3`.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.6: Measurements for **vectorized selection** operator on table “Lineorder” with **two predicates** `1≤discount≤3`.

Scale Factor	Selected OIDs
1	784 921
2	1 571 172
3	2 354 834
4	3 139 626
5	3 925 940
6	4 711 992
7	5 497 021
8	6 281 413
9	7 069 395
10	7 855 092

Table 6.4: Number of join partners for one sort-merge join on the “Lineorder” table in SSB query 1.1.

Scale Factor	Lineorder	Date
1	784 921	365
2	1 571 172	365
3	2 354 834	365
4	3 139 626	365
5	3 925 940	365
6	4 711 992	365
7	5 497 021	365
8	6 281 413	365
9	7 069 395	365
10	7 855 092	365

Table 6.5: Number of join partners for the hashjoin in SSB query 1.1.

Scale Factor	Lineorder
1	44 532
2	97 358
3	148 684
4	195 048
5	242 314
6	291 252
7	339 171
8	390 719
9	441 427
10	490 740

Table 6.6: Number of values to fetch from column “Lineorder”.“Revenue” for the fetchjoin.

6.2.3 Join Operators

Next, we consider a few join operators. In contrast to the filters, these are only implemented in a scalar code fashion. We will first investigate the performance of a sort-merge join for sorted inputs, then that of our fetchjoin for random access to BAT values, and finally a hash join implementation.

Sort-Merge Join

First, we look at a sort-merge join implementation, which we call *matchjoin* in our *AHEAD* prototype. This join is designed to take as input two sorted columns with. For base columns, this is given anyways, because their head type is VOID. Here, we use two iterators, one per input column, at the same time. Only one iterator is incremented at a time until the value at its position is greater or equal to the value at the other iterator's position. When the items match, the head of the first BAT and the tail of the second BAT are materialized as output and both iterators' positions are incremented. When instead an iterator's value is larger than the other's, that second iterator is incremented until its value in turn is greater or equal to the first iterator. For the SSB queries, we only need this operator when we selected OIDs from one column, using the σ operator, and then want to fetch only the matching values from another column of the same table. For instance, in SSB query 1.1, we select the values from “Lineorder” attribute “Discount” which satisfy the predicate $1 \leq \texttt{discount} \leq 3$. The selection operator only returns the respective (sorted) OIDs and because we need the discount later in the query again, we must also retrieve the actual values from the “Discount” column, which is achieved with the matchjoin operator. As we mentioned, it is in fact a sorted-merge join operator which can principally take any BAT variants. For our measurements from Figure 6.7, the number of OIDs to retrieve from the “Lineorder” table are listed in Table 6.4, again for scale factors 1 to 10. The picture here is like for the scalar selection operators, although in these measurements there seems to be clutter starting with scale factor 6. It makes sense that it behaves like the selection operator, because (1) it is like a selection over two columns in parallel, whereas not predicates are compared, but the values from the two columns, and (2) both have sequential access patterns. In contrast to the scalar selection, however, the Skylake IPC is much higher. This shows that it can efficiently issue the more operations required to increase the two column iterators used for the sort-merge behavior. In total, this operator leads to $\approx 10 \ldots 30\%$ runtime overhead for both measurement systems. The instruction overhead is around 20% and by that comparable to that from the scalar selection operator.

Hashjoin

The second join implementation which we investigate is a hash-based one. The hash join operator first builds a hash table for one input column and then probes the other columns against this hash table. The hash table is built using the first BAT's tail and then probes the second BAT's head, or the other way around, depending on the desired join order. As hash table implementation we use the Google implementation "sparsehash"[4]. Despite its name, the project offers implementations for both sparse and dense hash tables, whereas we only use the dense hash table variant. Figure 6.8 shows our measurements for the hash-based join of tables "Lineorder" and "Date" over attributes "orderdate" and "datekey", respectively. The hash table is built over the "Date" input column, since it is the much smaller dimension table. In contrast to the other operators, the hash join has many more random memory accesses. We see the difference especially for the Skylake system. The instruction overhead (Figure 6.8e) is 14.7% for continuous and 15.9% for recoding, on average. This is slightly lower than for the other scalar operators. The runtime overhead is 8.8% for the recoding variant and 11.1% for the normal hardened variant. Although both operate on the same physical data width, the recoding variant uses a smaller A and by that the actively used data domain is smaller. This may in total lead to the faster behavior of the hash table for the recoding hardened operator.

Fetchjoin

The final join is what we call *fetchjoin*, which realizes random access to a target BAT. As can be seen in Table 6.2, the fetchjoin takes a first BAT with a VOID/OID mapping, which denotes the OIDs of the values to fetch from the second BAT. Furthermore, we restricted for that join the second BAT to have a head sub-column of type VOID. This is to enforce at source code level that the second BAT is sorted and is used in the SSB queries to fetch values from base columns. The results in Figure 6.9 are taken from SSB query 2.1, where values are fetched from column "Revenue" of table "Lineorder". The numbers of values per scale factor, which the fetchjoin retrieves from column "Lineorder"."Revenue" are shown in Table 6.6. For each scale factor, the ratio of the retrieved values from the "Lineorder" table is $\approx$ 8‰. As the graphs from Figures 6.9a to 6.9d show, the runtime overhead of $\approx 12.5\%$ is again on the same level as the previous scalar operators. However, the instruction overhead in terms of additionally retired instructions (Figures 6.9e and 6.9f) is considerably lower than for the other scalar operators, with averages of 11.5% for continuous and 13.5% for recoding, for the Skylake system. On the Xeon Phi system, this overhead is on average 5.2% and 5.6% for continuous and recoding, respectively. Additionally, the IPC is much lower, with 0.5 and 0.3 for Skylake and Xeon Phi, respectively.

[4] See `https://github.com/sparsehash/sparsehash`

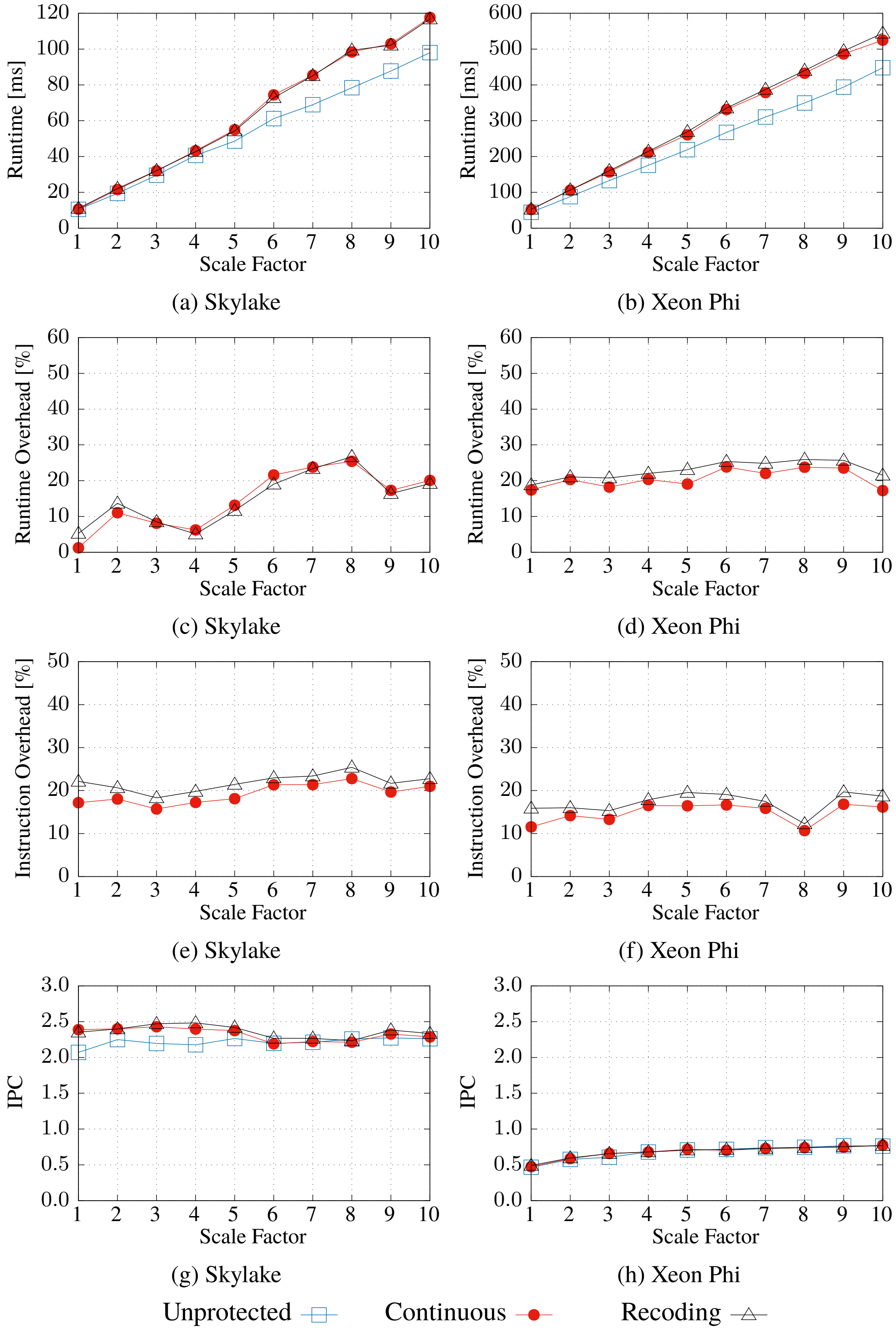
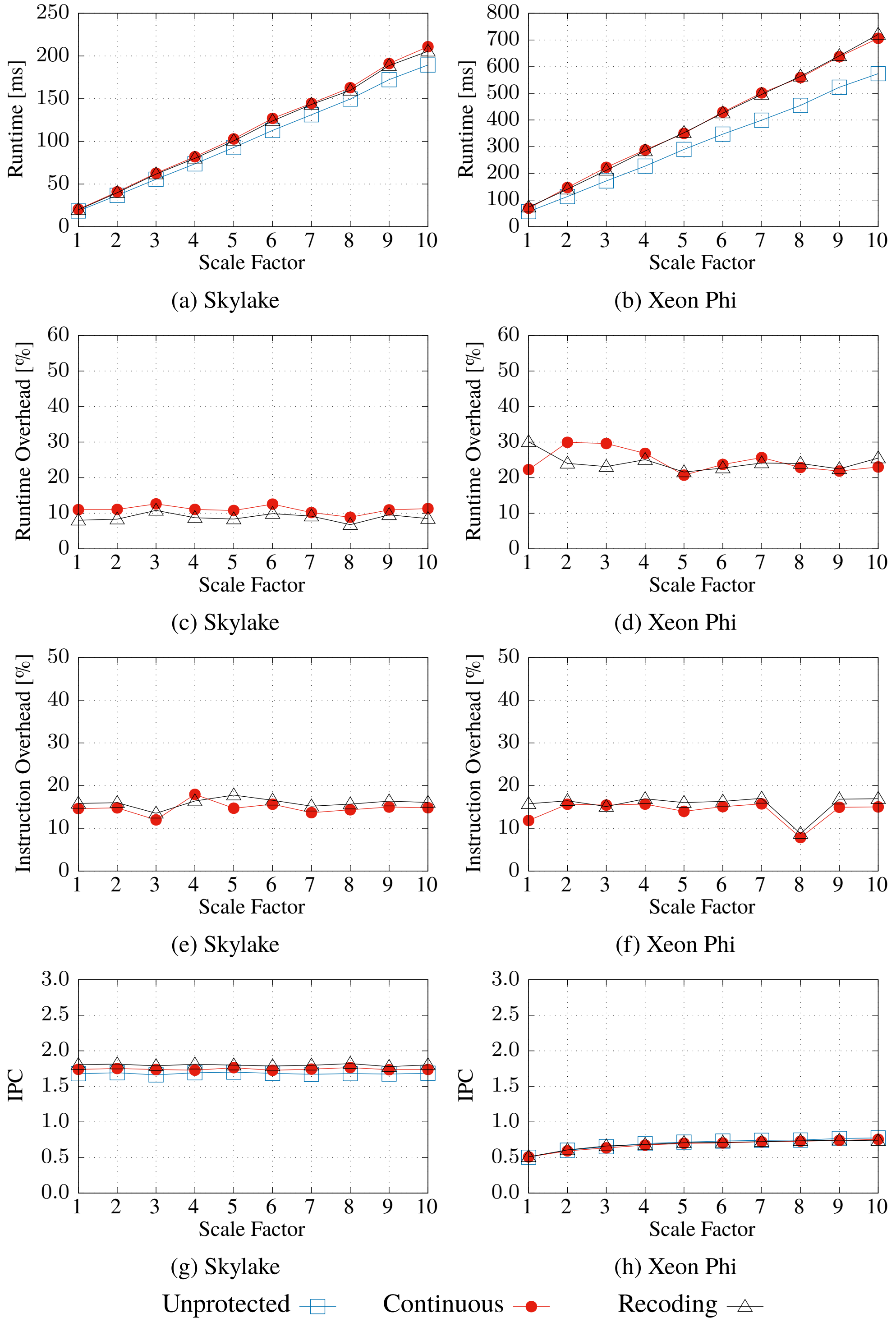


Figure 6.7: Measurements for **scalar sort-merge join** operator retrieving values for column “Discount” from table “Lineorder”.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

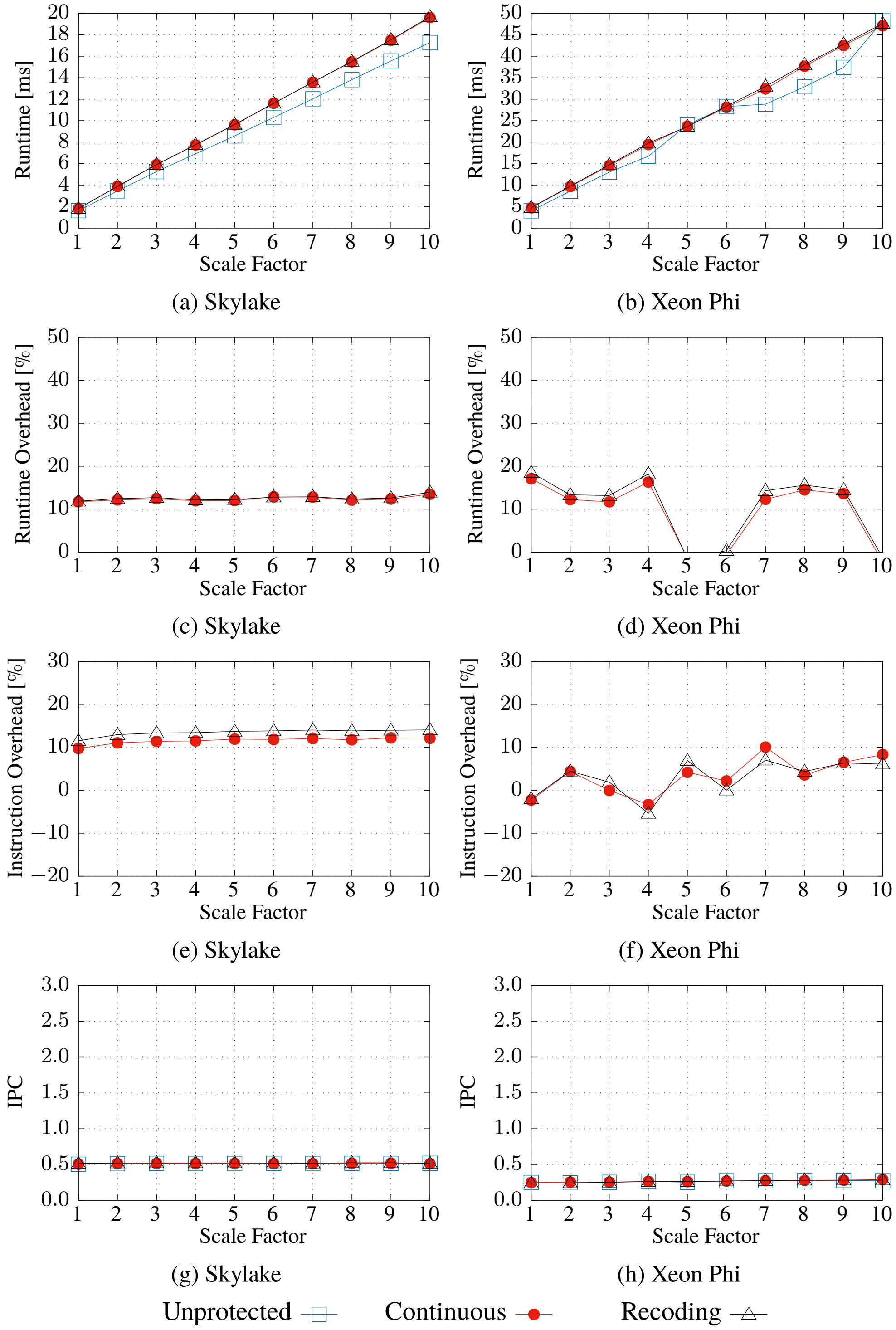
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.8: Measurements for **scalar hashjoin** operator joining tables “Lineorder” and “Date”.



(a) Skylake (b) Xeon Phi

(c) Skylake (d) Xeon Phi

(e) Skylake (f) Xeon Phi

(g) Skylake (h) Xeon Phi

Figure 6.9: Measurements for **scalar fetchjoin** operator retrieving values from column “Revenue” of table “Lineorder”.

Scale Factor	Column Cardinality	# Groups Unary	Binary
1	44 532	7	280
2	97 358	7	280
3	148 684	7	280
4	195 048	7	280
5	242 314	7	280
6	291 252	7	280
7	339 171	7	280
8	390 719	7	280
9	441 427	7	280
10	490 740	7	280

Table 6.7: Cardinality of the columns for grouping in SSB query 2.1.

Scale Factor	Cardinality
1	118 598
2	238 451
3	357 647
4	477 200
5	596 361
6	716 148
7	835 037
8	953 544
9	1 072 992
10	1 193 001

Table 6.8: Cardinality of the columns for multiply-and-sum aggregation in SSB query 1.1

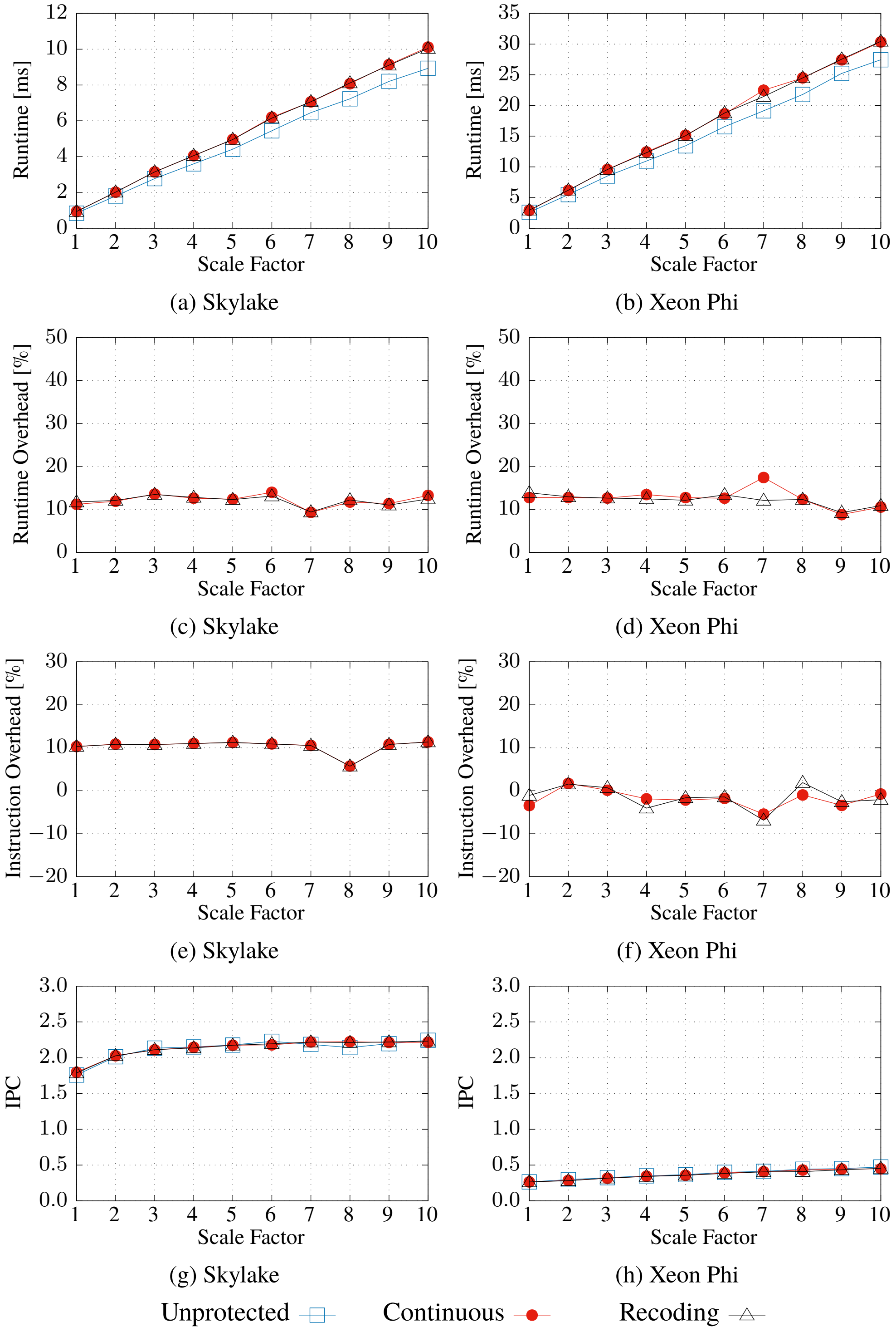
6.2.4 Grouping and Aggregation

Next, we consider grouping and aggregation operators. First, we measure a unary group-by operator, where intermediate results of SSB query 2.1 are grouped by the "Year" attribute from table "Date". This one uses a single hash table to test whether a value was already present, where we again use Google's sparsehash project implementation. The group by operators actually return two BATs, where the first is a VOID-to-OID mapping from the original BAT's entries into the second BAT which stores the groups in a VOID-to-value mapping. Table 6.7 lists the cardinalities for the input BAT, as well as the number of groups it generates in the third column. As we can see, there are seven distinct values, i.e. groups, for all scale factors. Figure 6.10 shows the results, where we again see a low runtime overhead of around 12% on average for both hardened variants, compared to the unencoded operator, as seen in Figures 6.10a to 6.10d. The runtime overhead is approximately 12% on average for both measurement systems. Also, the instruction overhead itself is quite low with 10% on average for the Skylake system (Figure 6.10e) and with virtually no overhead for the Xeon Phi (Figure 6.10f). Regarding the IPC neither of the encoded or unencoded variants has an advantage for this operator (Figures 6.10g and 6.10h).

Next, we consider a binary group-by operator, which takes as an additional parameter an existing group mapping, e.g. from the unary group-by operator, or another binary group-by. By that, multiple group-by operations can be chained such that we only need these two group-by operators. An extra constraint is that the input BAT must have the same cardinality as the existing group mapping. In Table 6.7, the fourth column lists the number of generated groups, where we again see that for all scale factors, 280 groups are generated. The additional mapping that his operator takes comes from the previously described unary group-by. The measurements are shown in Figure 6.11, which belong to the binary group-by operator which is executed right after the unary in SSB query 2.1, i.e. after the one we previously discussed. We first see, that the runtimes, albeit on the same cardinality input, are several times higher than those from the unary group by. This can attributed to the added access to the existing grouping. Here both the runtime overheads (Figures 6.10c and 6.10d) as well as the instruction overheads (Figures 6.10e and 6.10f) are marginal for both systems and also the IPC is virtually the same between all three operator variants.

Next, we show measurements for a sum aggregation operator in Figure 6.12. This operator is also taken from SSB query 2.1, where it computes a sum per grouped elements in a column. In particular, it is executed after the previously discussed binary group-by operator, respecting its computed groups. As it works on the groupings, we only implemented a scalar variant, like the group-by operators themselves. We see for the Skylake system in Figures 6.12a and 6.12c, that the runtimes of the hardened operators are only marginally slower than for the unprotected aggregation operator, with 3.5% and 4.3% on average for continuous and recoding, respectively. The Xeon Phi system, in contrast, shows runtime overheads of respectively 16.1% and 15.6% on average. The instruction overhead is 17.7% for Skylake for both encoded variants, and for the Xeon Phi 4.5% and 3.9% on average for continuous and recoding, respectively. The IPC levels are also again very different between the two systems. For the Skylake system the averages are 1.744, 1.996, and 1.98 for Unprotected, Continuous, and Recoding, respectively. For the Xeon Phi, they are much lower with 0.290, 0.301, and 0.301, respectively.

Finally, we present runtimes for one last aggregation operator, which is used in SSB query 1.1, where it iterates over two input columns and computes the product of the value (tail) from each column and in parallel sums up all the products. In particular, the operator takes filtered columns “Extendedprice” and “Discount” from table “Lineorder”, where the cardinalities of both inputs are listed in Table 6.8 for the ten scale factors, which constitute $\approx 2\%$ of the base table’s total number of tuples. We use this as a shorthand, instead of implementing two separate operators, for which we provide both a scalar and a vectorized variant. The measurements for the scalar variant are shown in Figure 6.13. Here, the runtime overhead for Continuous and Recoding is again quite moderate, with averages of 4.2% and 4.0% for Skylake, respectively, and 11% and 10.4% for Xeon Phi, respectively (Figures 6.13c and 6.13d). The instruction overhead is again higher for Skylake, with 15.3% and 15.6% on average, respectively, whereas this is only 6.5% and 6.6% for the Xeon Phi on average (Figures 6.13e and 6.13f). Again, the Skylake can much better handle the additional work for error detection, which becomes evident by the higher IPC of the hardened operators running on Skylake, which are 2.34 in contrast to 2.10 for Unencoded (Figure 6.13g). The measurements for the SSE variant are shown in Figure 6.14 and shed a very different light on the hardened operator performance. The runtimes differ *extremely* for both systems, with averages around 400% and 1000% for Skylake (Figure 6.14c) and Xeon Phi (Figure 6.14d), respectively. Interestingly, the instruction overhead is only much higher for the Skylake system, with 211% on average for both hardened operators (Figure 6.14e), whereas on the Xeon Phi system the instruction overhead is only 25% on average. The IPC numbers look again more like the other operators, whereas for Skylake the encoded operators can again mostly retire more instructions than their Unprotected counterpart. The extreme runtime difference can be attributed to the simplicity of the operator. The operator does only produce a single result (the sum) and, therefore, the CPUs can much better schedule the actual operations. In contrast to the column filter operators from Figures 6.4 and 6.6, here no intermediate materialization is required. Additionally, the Unprotected operator only has the loop condition to test which iterates over the column values, which means there are no branches in the inner loop. In contrast, the hardened operators introduce an additional multiplication, comparison and branch (test for corrupt values) for each input column. Depending on the CPU’s ability of branch prediction, the additional branch greatly affects the hardened operators’ runtimes. On top of that, the hardened operators can again only process half as many values per vector instruction as the Unprotected variant. However, considering the absolute runtimes, this aggregation operator is one of the shortest running ones, compared to any of the other presented operators.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.10: Measurements for **scalar unary group-by** operator on a single column.

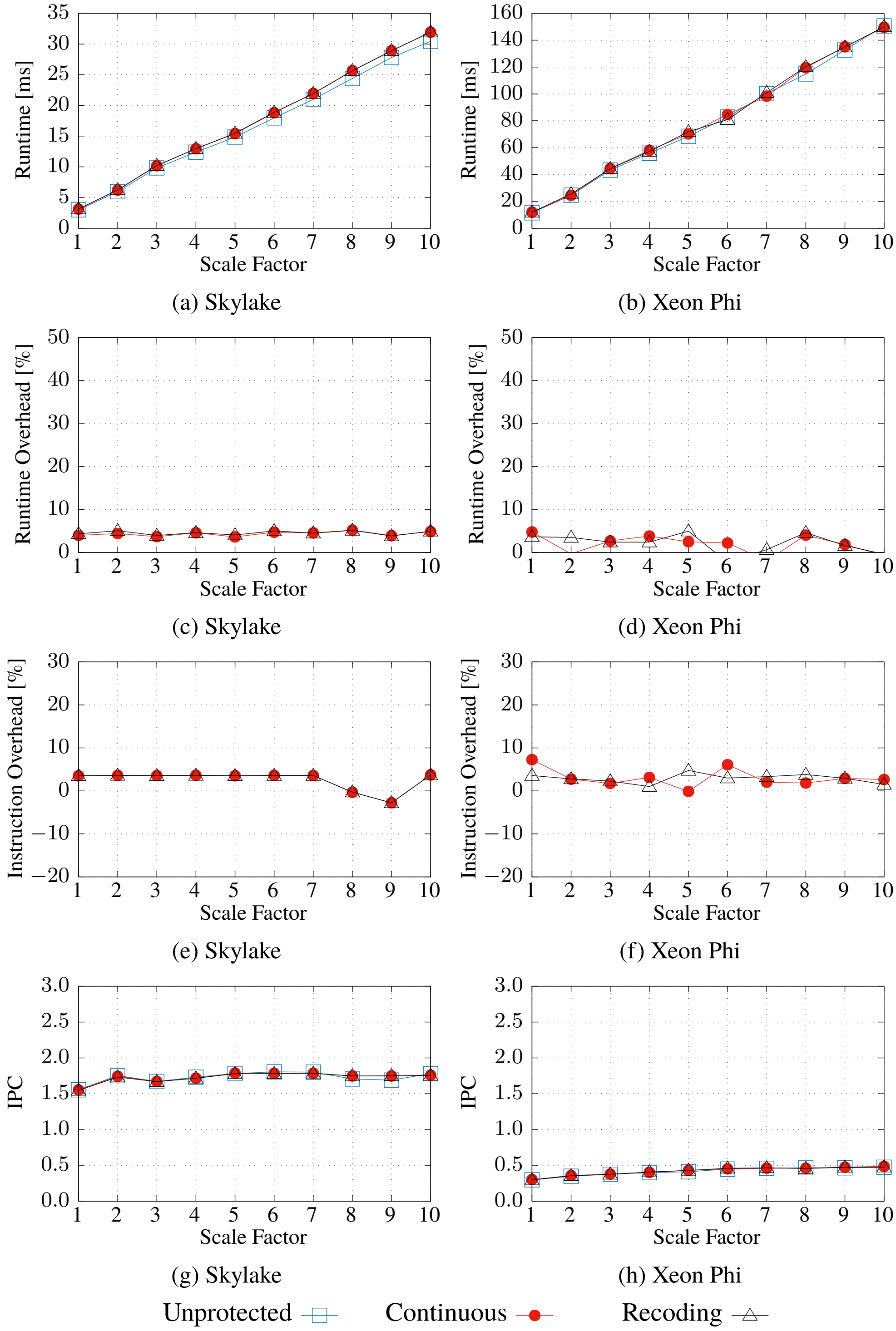
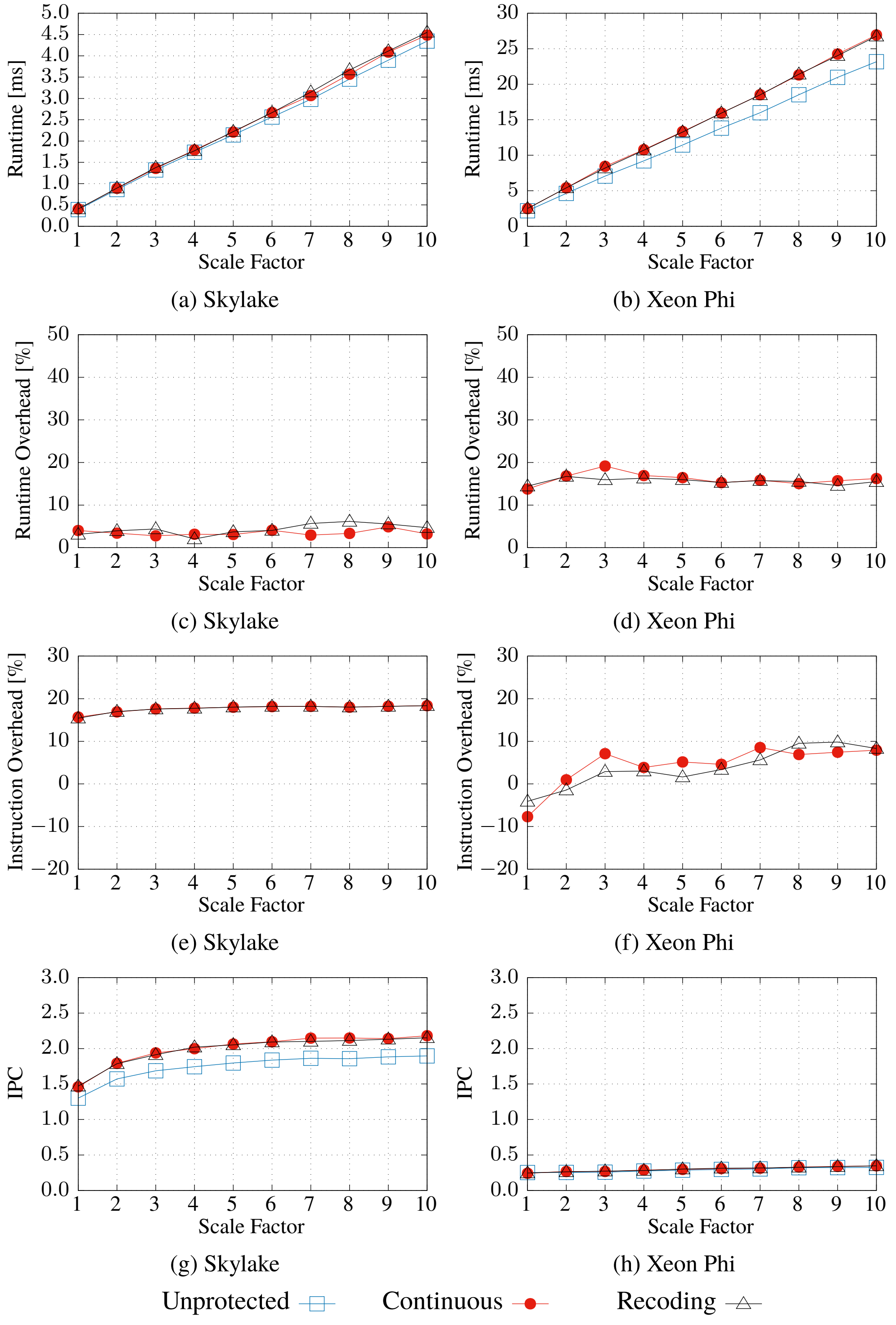


Figure 6.11: Measurements for a **scalar binary group-by** operator on a column with an additional, existing grouping.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

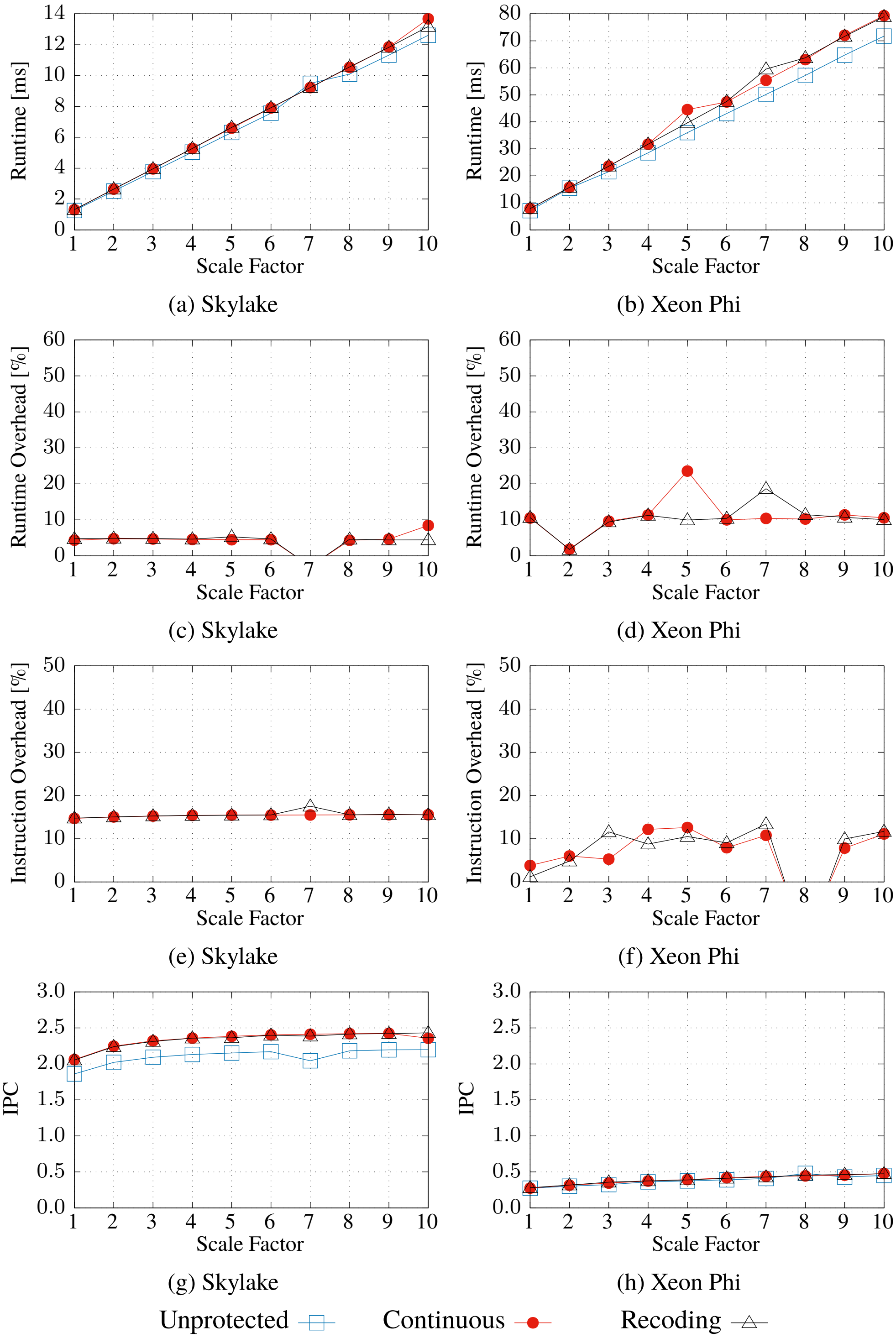
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.12: Measurements for **scalar sum aggregation** operator on a **grouped** column.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

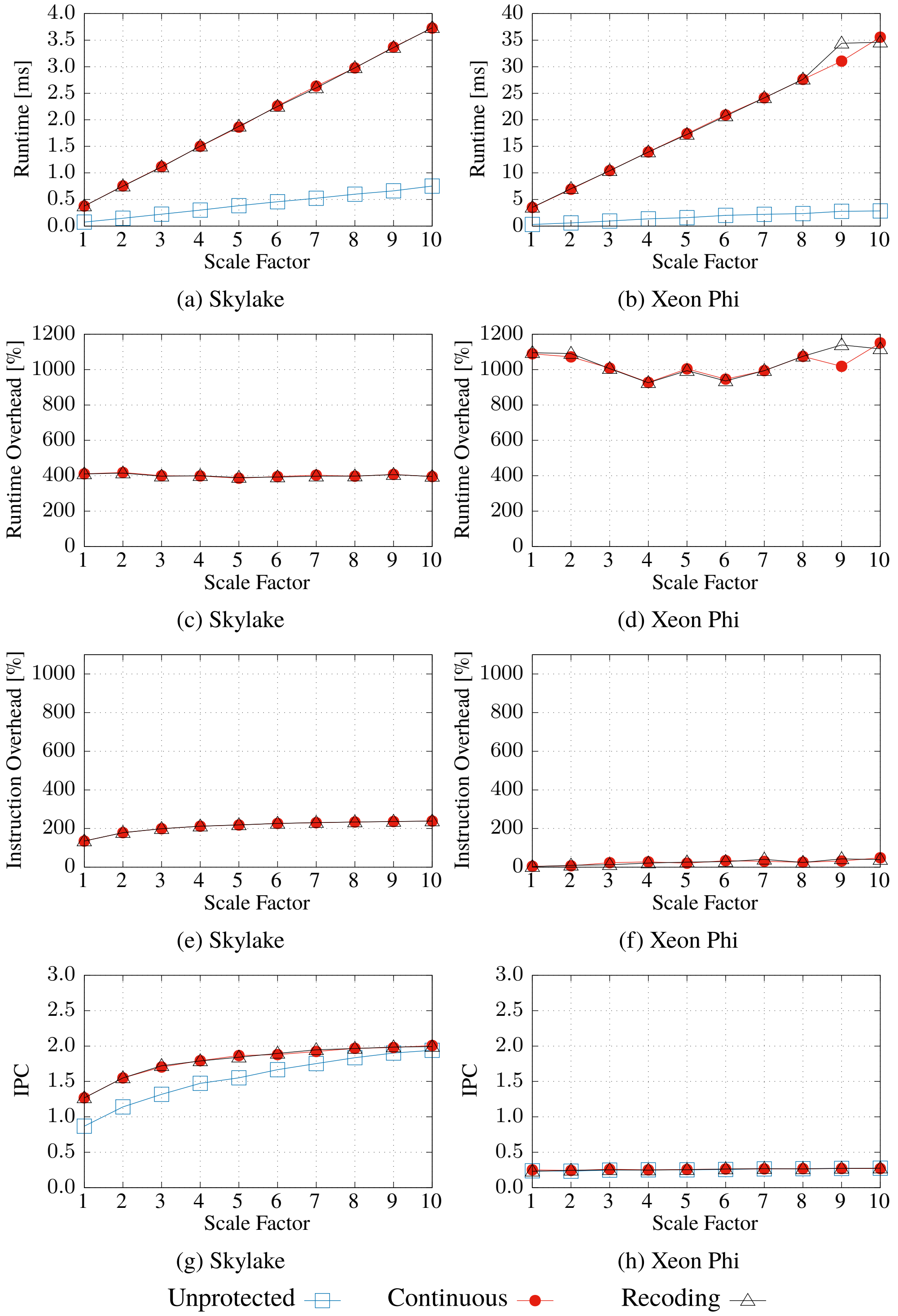
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

Figure 6.13: Measurements for a **scalar aggregation** operator, which **multiplies** 2 columns and computes the **sum** of the products.



(a) Skylake

(b) Xeon Phi

(c) Skylake

(d) Xeon Phi

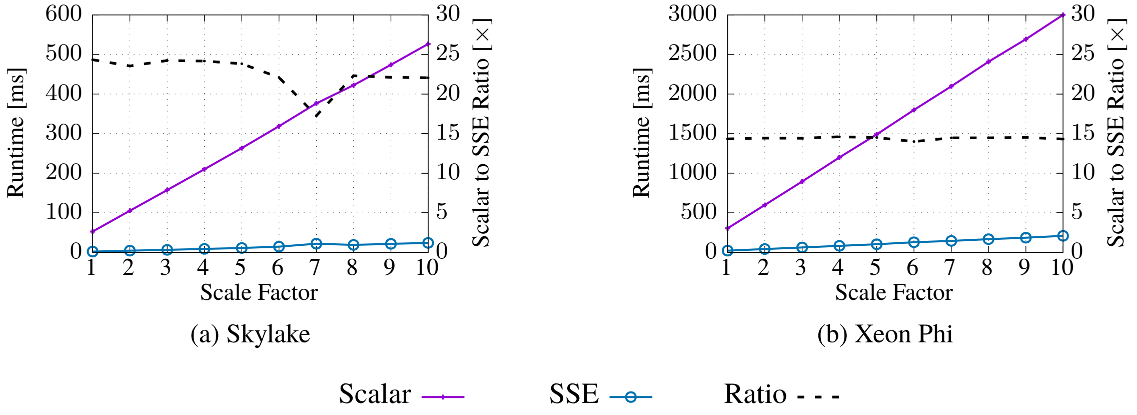
(e) Skylake

(f) Xeon Phi

(g) Skylake

(h) Xeon Phi

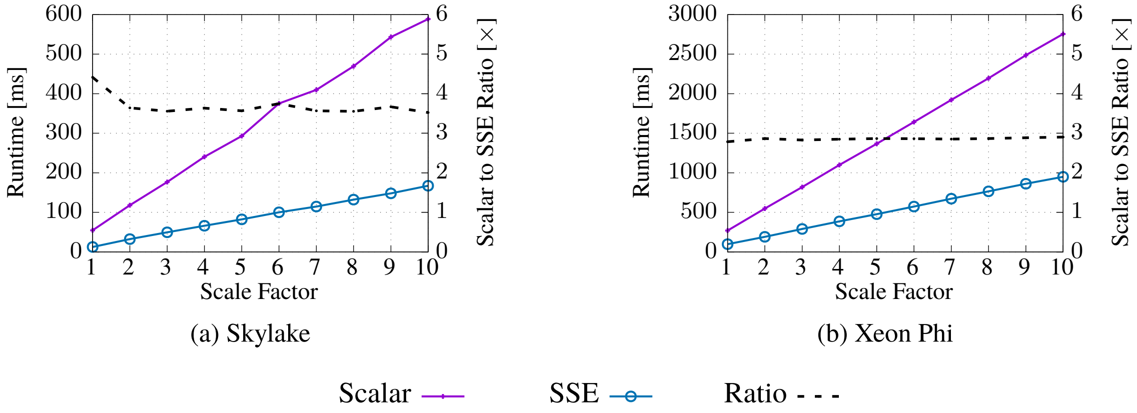
Figure 6.14: Measurements for a **vectorized aggregation** operator, which **multiplies** 2 columns and computes the **sum** of the products.



(a) Skylake

(b) Xeon Phi

Figure 6.15: Δ operator on encoded `AN_tiny` (16-bit)



(a) Skylake

(b) Xeon Phi

Figure 6.16: Δ operator on encoded `AN_int` (64-bit)

6.2.5 Delta Operator

The final operator we discuss in detail is the Δ operator, which is only used by the Early and Late Detection opportunities. A crucial difference to the Since it has no unencoded counterpart, we will only show in how far it benefits from vectorization. We consider small (16-bit `AN_tiny`) and large (64-bit `AN_int`) encoded data (cf. Table 6.1). From SSB query 1.1, we present the measurements for columns “discount” (`AN_tiny`) in Figure 6.15 and “orderdate” (`AN_int`) in Figure 6.16. The left y-axis shows the absolute runtimes for scalar and SSE execution, while the right y-axis depicts the ratio of scalar to SSE runtime $^{\text{runtime scalar}}/_{\text{runtime SSE}}$. In Figure 6.15 we see that for very small encoded data, the vectorization ratio is mostly above 22× (Figure 6.15a) and 14× (Figure 6.15b). This is even greater than the ratio of 16-bit data elements per SSE4.2 register, which is 8. A similar behavior can be seen for the larger `AN_int` case in Figure 6.16, where the ratio is above 3.5× (Figure 6.16a) and around 2.9× (Figure 6.16b). This is again greater than the ratio of 64-bit data elements per SSE4.2 register, which is 2. This in total shows that the Δ operator benefits greatly from vectorization and this is another proof for the benefits of our coding improvements presented in Section 3.5.

6.2.6 Summary and Conclusions

In this section, we presented the runtime behaviors of a wide range of physical operators, which were taken from actual SSB queries. The choice of operators included selection operators on one and two predicates, three different joins, as well as two grouping and two aggregation operators. We showed detailed measurements for absolute runtimes, runtime overheads, overheads of retired instructions, as well as instructions per cycle (IPC). For all of the scalar operator variants, the runtime overheads of Continuous and Recoding were in the range between $3.5 \dots 14.9\%$ for the Skylake system and between $1.9 \dots 24.7\%$ for the Xeon Phi system. The retired instruction overheads were between $2.6 \dots 21.9\%$ for both systems, which is more or less in the same range. However, we saw that the more powerful Skylake cores could typically better schedule the additional instructions for error detection, resulting mostly in lower runtime overheads than for the Xeon Phi. This means, for the Skylake system, the instruction overhead was typically larger than the actual runtime overhead. On the Xeon Phi system, it was mostly the other way around – the runtime overhead was mostly higher than the instruction overhead. Since the instruction overhead is also typically smaller than for the Skylake system, the code generated for the Xeon Phi system requires more instructions in general. Regarding vectorized performance, the runtime overheads increase for the selection operators to $44 \dots 53\%$ for Skylake and to $33 \dots 35\%$ for Xeon Phi. For the last aggregation operator, the runtime overheads even to around 400% and 1000% for Skylake and Phi, respectively. One issue is that the hardened variants can only process half as many values per vector instruction due to the larger data widths. This last operator, however, is purely memory read and compute intensive, where there are little to no stalling cycles where the CPU can hide the error detection instructions in between. By that, the proportion of the instruction overhead is much higher, too, even 210% for Skylake. Therefore, when there is more work to do in general, even for the selection operators, where more data is written back to memory, error detection has less impact on the operator runtime. Furthermore, we can see that recoding does introduce virtually no additional overhead to error detection. Finally, we investigated the performance of the Δ operator, which is used for the Early and Late Detection opportunities. As we showed, it gains extremely well from vectorization and the scalar to SSE runtime ratio was even larger than the data elements per vector register ratio.

6.3 STAR SCHEMA BENCHMARK QUERIES

In the previous section, we dissected the overheads of individual operators, where we compared the unprotected variants against hardened and recoding ones. In this section, we will consider whole queries, namely those from the star schema benchmark [131]. It consists of 13 queries in total, with a mixture of selections on one or two predicates, some table joins, as well aggregations and groupings. For a detailed description of the benchmark suite, we would like to direct the reader to [131]. We measured runtimes for scale factors one to ten, for which the cardinalities of the tables are shown in Table 6.3. Also, all measurements were taken for single-threaded execution. In the following, we will present measurements for the query runtimes, then we discuss the impact of vectorization on whole query runtimes, and finally we examine the storage overhead. In the following, we will first present the runtimes for the complete queries, divided into scalar and SSE exeuction on their own. Then, secondly, we will discuss the improvements by vectorization, first for only the vectorizable operators and then for the whole queries. This goes further than in the last section, where we investigated the operators individually. Third and finally, we will discuss the storage overhead for base data and intermediate results.

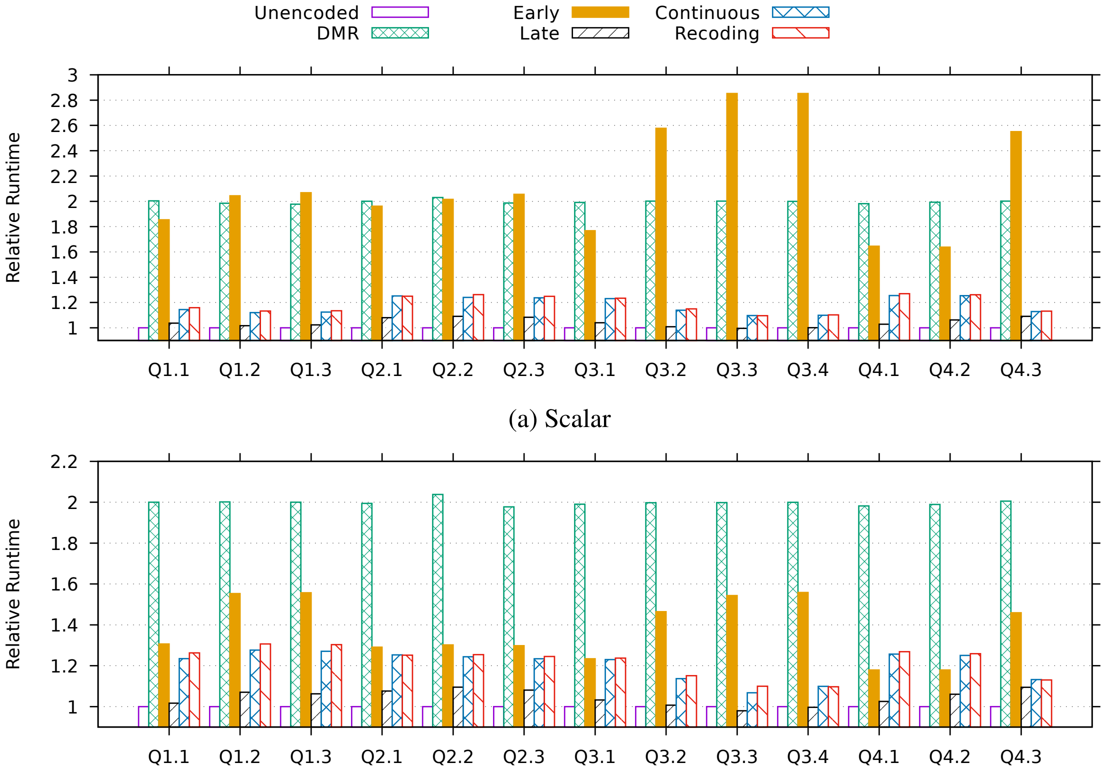
6.3.1 Query Runtimes

We measured runtimes for both systems and both scalar and vectorized execution. We compare six variants, namely the original, unprotected variant, a double modular redundancy (DMR) variant, Early and Late Onetime Detection, as well as Continuous Detection and its Recoding variant. DMR stores all data twice, then executes a query two times successively, and afterwards compares the results of both queries, to test whether some error occurred. We compare against a DMR approach, because this is state of the art for so many other generic transient error detection approaches, from which we introduced many in Chapter 2. It is the smallest variant for n-modular redundancy and here we execute the two parts of the query *sequentially*. While this could be done in parallel as well, it would then cost 2× computing resources. However, for a better visualization of the impact of DMR we chose sequential execution. After we previously discussed the individual operator runtimes of unprotected and AN hardened variants individually, we now see in how far these impact the whole query runtimes.

Figure 6.17 shows the query runtimes for the Skylake system and Table 6.9 lists the actual numbers, as well as minimum, maximum, and average values. All values are relative to the Unprotected case and averages over scale factors $1 \ldots 10$ and for each scale factor we executed 10 runs. In total, this accumulates to 100 runs per value. For the SSB measurements in our *AHEAD* prototype, we do not use unrolling. As we have seen in the microbenchmarks in Sections 3.4.2 and 3.5.4, loop unrolling can gain some benefit, but is yet another optimization dimension which we do not consider now.

We will first consider the scalar execution for the Skylake system. DMR approximately shows the 2× query runtime which we would expect. For Early Onetime, we see now the impact of the early materialization, because the detect-and-decode operator is the only addition to the unprotected case here. Early is between $59\% \ldots 178\%$ slower than the Unprotected baseline for scalar execution. With an average overhead of 108%, Early Onetime is even slower than DMR. For each of the queries 3.2, 3.3, and 3.4 the proportion of all Δ operators accumulates to more than 64% of the total query runtime, which is why these queries have exceptionally high runtime overheads for Early Onetime. In contrast, for Late Onetime we insert Δ just before the last query operator. This is always a group-by or aggregation operator, so we run error detection on more values than just at the very end of each query. Here, the impact is almost negligible, with a runtime overhead of just 1%, on average. For the Continuous and Recoding variants, the overheads are quite low and range between $6 \ldots 23\%$, with averages of 14% and 15%, respectively. Interestingly, the picture is almost the same for the vectorized execution, except for Early Onetime. As we just showed above, the Δ operator at the very beginning gains so much from vectorization that the overhead drops to 14%, 51%, and 33% for minimum, maximum, and on average, respectively. We can very well see the effect for the already mentioned queries 3.2 to 3.4, where the runtime overheads drop to $42\% \ldots 51\%$. For Continuous and Recoding, we had seen for the operators from the last section that the runtime overhead in general increased for the vectorized operators (column scan, and aggregate). However, the runtime proportion of these operators in comparison to that of the non-vectorized operators is only marginal. By that, the total runtimes of Continuous and Recoding do not suffer as much as the operator measurements suggested. Now, they show average runtime overheads of 17% and 18%, which is an increase of 3% for both.

Figure 6.18 shows the runtimes for the Xeon Phi system and the actual numbers are listed in Table 6.10. Here again the relations are much like for the Skylake system. The DMR variant as about the 2× overhead for both scalar and SSE execution. Early Onetime is again slower than DMR for the scalar case and catches up quite a lot when using vectorization. As we saw in the microbenchmarks before, the Xeon Phi system is much slower than the Skylake system regarding single thread performance and is built from much older cores in principle than Skylake. By that, the vectorization gains are not as high as for the Skylake system and for the former we still have 47% overhead on average, while the latter average went down to 33%. The overheads of Late Onetime are also almost negligible and as for Early Onetime they are slightly higher on average than on the Skylake system. In contrast, Continuous and Recoding have even slightly less overhead compared to the Skylake system in the respective execution modes, with 11% for the scalar case and 12% when running with SSE. On the one hand, this is a bit surprising as Early and Late were slower in comparison. On the other hand, the absolute runtimes are larger than on the Skylake system, e.g., for query 1.1 the runtimes for system 2 are on average $4.4\times$ and $5.6\times$ slower than system 1 across all scale factors.



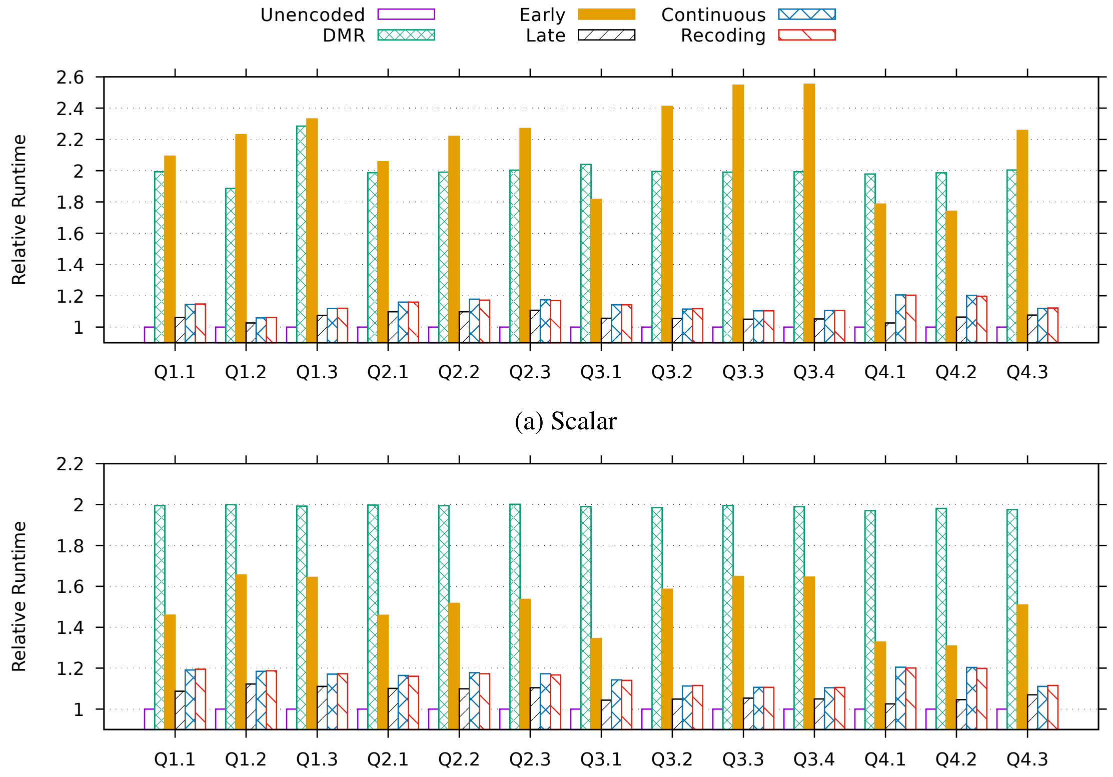
(a) Scalar

(b) SSE

Figure 6.17: Runtimes of SSB queries on the Skylake system.

Query	Scalar					SSE				
	DMR	Early	Late	Cont.	Recod.	DMR	Early	Late	Cont.	Recod.
Q1.1	1.94	1.80	1.01	1.11	1.12	1.94	1.26	0.99	1.20	1.23
Q1.2	1.92	1.98	0.99	1.09	1.10	1.94	1.50	1.04	1.24	1.27
Q1.3	1.92	2.01	0.99	1.09	1.10	1.94	1.51	1.03	1.23	1.26
Q2.1	1.94	1.90	1.05	1.21	1.21	1.93	1.25	1.04	1.22	1.21
Q2.2	1.97	1.96	1.06	1.20	1.23	1.98	1.26	1.06	1.21	1.22
Q2.3	1.92	1.99	1.05	1.20	1.21	1.92	1.25	1.05	1.20	1.21
Q3.1	1.93	1.72	1.01	1.20	1.20	1.93	1.20	1.00	1.19	1.20
Q3.2	1.94	2.50	0.98	1.11	1.12	1.94	1.42	0.98	1.10	1.12
Q3.3	1.94	2.78	0.97	1.06	1.06	1.94	1.50	0.95	1.04	1.07
Q3.4	1.94	2.78	0.97	1.07	1.07	1.94	1.51	0.97	1.07	1.06
Q4.1	1.93	1.59	1.00	1.22	1.23	1.93	1.14	1.00	1.22	1.23
Q4.2	1.94	1.59	1.03	1.22	1.23	1.93	1.14	1.03	1.21	1.22
Q4.3	1.94	2.48	1.06	1.10	1.10	1.95	1.41	1.06	1.10	1.10
Minimum	1.92	1.59	0.97	1.06	1.06	1.92	1.14	0.95	1.04	1.06
Maximum	1.97	2.78	1.06	1.22	1.23	1.98	1.51	1.06	1.24	1.27
Average	1.94	2.08	1.01	1.14	1.15	1.94	1.33	1.01	1.17	1.18

Table 6.9: Runtime numbers of SSB queries on the Skylake system compared to the Unprotected baseline.



(a) Scalar

(b) SSE

Figure 6.18: Runtimes of SSB queries on the Xeon Phi system.

Query	Scalar					SSE				
	DMR	Early	Late	Cont.	Recod.	DMR	Early	Late	Cont.	Recod.
Q1.1	1.93	2.04	1.03	1.11	1.11	1.93	1.41	1.05	1.16	1.16
Q1.2	1.83	2.17	0.99	1.03	1.03	1.94	1.61	1.09	1.15	1.15
Q1.3	2.23	2.27	1.04	1.09	1.09	1.93	1.60	1.08	1.14	1.14
Q2.1	1.93	2.00	1.06	1.13	1.13	1.94	1.41	1.07	1.13	1.13
Q2.2	1.93	2.16	1.06	1.15	1.14	1.93	1.47	1.06	1.14	1.14
Q2.3	1.94	2.21	1.07	1.14	1.14	1.94	1.49	1.07	1.14	1.13
Q3.1	1.98	1.77	1.02	1.11	1.11	1.93	1.30	1.01	1.11	1.11
Q3.2	1.93	2.36	1.02	1.08	1.09	1.93	1.54	1.02	1.08	1.08
Q3.3	1.93	2.50	1.02	1.07	1.07	1.94	1.61	1.02	1.07	1.08
Q3.4	1.93	2.50	1.02	1.07	1.07	1.93	1.60	1.02	1.07	1.07
Q4.1	1.92	1.74	0.99	1.17	1.17	1.91	1.29	0.99	1.17	1.17
Q4.2	1.93	1.69	1.03	1.17	1.16	1.92	1.27	1.01	1.17	1.16
Q4.3	1.94	2.20	1.04	1.09	1.09	1.92	1.46	1.04	1.08	1.08
Minimum	1.83	1.69	0.99	1.03	1.03	1.91	1.27	0.99	1.07	1.07
Maximum	2.23	2.50	1.07	1.17	1.17	1.94	1.61	1.09	1.17	1.17
Average	1.95	2.12	1.03	1.11	1.11	1.93	1.47	1.04	1.12	1.12

Table 6.10: Runtime numbers of SSB queries on the Xeon Phi system compared to the Unprotected baseline.

System	Type	Unprotected	DMR	Early	Late	Continuous	Recoding
Skylake	Scalar Proportion	0.56	0.56	0.76	0.58	0.54	0.53
	SSE Proportion	0.10	0.10	0.30	0.11	0.12	0.12
	Difference	0.46	0.46	0.46	0.47	0.42	0.41
	Ratio $^{\text{scalar}}/_{\text{SSE}}$	12.0	11.9	7.6	11.4	8.6	8.6
Xeon Phi	Scalar Proportion	0.53	0.52	0.79	0.54	0.50	0.50
	SSE Proportion	0.18	0.18	0.42	0.21	0.20	0.20
	Difference	0.35	0.34	0.37	0.33	0.30	0.30
	Ratio $^{\text{scalar}}/_{\text{SSE}}$	5.0	5.0	5.0	4.3	4.1	4.1

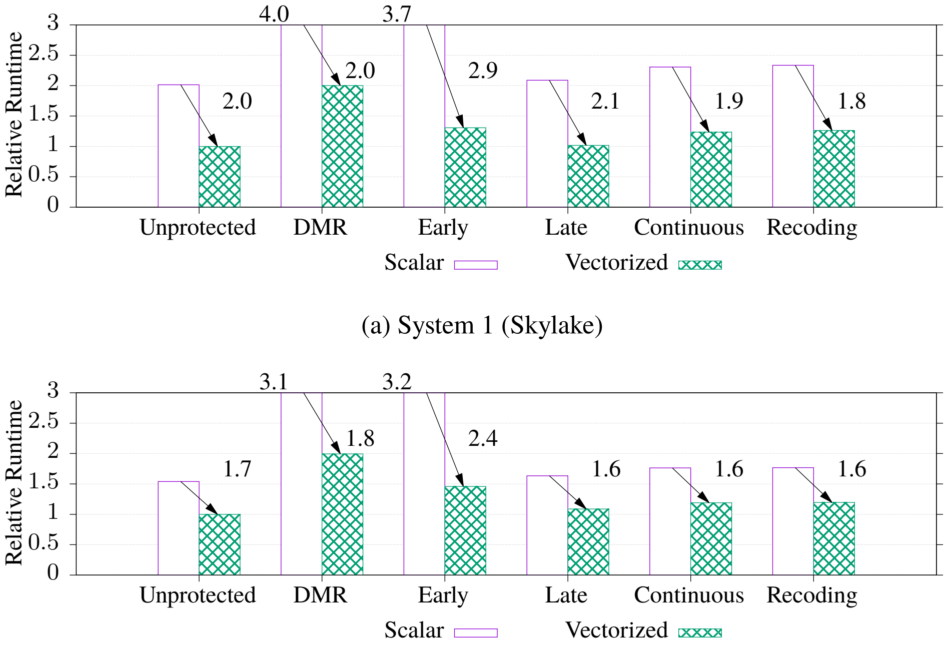
Table 6.11: Average runtime proportion and ratio of the vectorizable operators for SSB query 1.1.

6.3.2 Improvements Through Vectorization

We will now examine the impact of vectorization on the query runtimes and compare the benefits between the measured execution variants. From the 13 SSB queries, it turns out that only queries 1.1 to 1.3 considerably benefit from vectorization. This is, because the queries are dominated by the selection (scan) operators and also the Δ operators in case of Early and Late Detection. As we showed in Table 6.2, we vectorized the Δ, select, arithmetic and summation operators. We give an example for SSB query 1.1, where there are three selection operators and one aggregation operator which computes the sum of the products of two columns. In addition, Early requires six Δ operators to decode the respective six base columns and Late requires two Δ operators to decode the two intermediate columns which are the input columns for the final aggregation operator. The proportion of those vectorizable operators with regard to the total query runtime is shown in Table 6.11. There, we see that in the scalar cases they amount to at least 50% of the total query runtime for both systems. Using SSE 4.2 vectorization, except for Early the proportion drops to $10\% \ldots 12\%$ for the Skylake system, and to $18\% \ldots 21\%$ for the Xeon Phi system. For Early Detection, the proportions are reduced from 76% to 30% on the Skylake system and from 79% down to 42% for the Xeon Phi system. This means that the proportion is reduced by $41\% \ldots 47\%$ for Skylake and by $30\% \ldots 37\%$ for Xeon Phi, for query 1.1. The individual operators' improvements through vectorization were discussed in detail above, in Section 6.2.

Next to the proportion of the total query runtime, Table 6.11 also lists the ratio between the scalar and SSE runtime of those vectorizable operators we just discussed, for SSB query 1.1. This is the average ratio of scalar runtimes divided by SSE runtimes, arithmetically averaged over all scale factors. The Unprotected and DMR variants benefit the most with ratios of 12 (Skylake) and 5 (Phi). The Continuous and Recoding variants benefit slightly less with ratios of 8.6 (Skylake) and 4.1 (Phi). This shows that, while the AN coded operator variants do scale very well, they still scale slightly worse than the baseline operator variants without error coding. Furthermore, we see that for the (weaker) Xeon Phi system, the differences between unencoded and AN coded operator variants are much smaller than for the (stronger) Skylake system.

The impact on the total runtime of query 1.1 is visualized in Figure 6.19, again with averages over all scale factors from 1 to 10. There, for each variant are shown two bars, which represent the scalar and vectorized (SSE) runtimes, all relative to the fastest one (vectorized Unprotected). The *scalar* runtimes of DMR and Early are pretty large so we capped the bars and the relative numbers are given above each respective bar. The arrows between each scalar and vectorized bar and the accompanying numbers denote the vectorization improvement for variant itself. This is again the ratio scalar runtime divided by SSE runtime. For the Skylake system, the total query runtimes



(a) System 1 (Skylake)

(b) System 2 (Xeon Phi)

Figure 6.19: Total query runtime improvements through vectorization for SSB query 1.1. Runtimes are given relative to Unprotected SSE execution. Arrows denote the respective improvement per detection variant as the ratio between scalar and vectorized runtime.

improve by almost the same factors between $1.8 \ldots 2.1$, except for Early, where the Δ operator improvements have an even greater impact leading to a ratio of 2.9. The picture of the Xeon Phi system in terms of improvements is pretty close with a small offset. The variants improve by factors between $1.6 \ldots 1.8$ wheras Early has again a higher ratio of now 2.4.

6.3.3 Storage Overhead

Next to runtime, memory consumption is another important factor regarding efficiency. In Sections 3.2.3 and 3.4.2 we discussed in detail how the choice of AN coding parameter A influences the storage overhead. In particular, with Table 3.6 we provided an easy way to determine the smallest A which provides a guaranteed minimal detectable bit flip weight (η) for a given data width. In the following, we will first show the storage requirements per data type based on η. Afterwards, we will measure the impact of η on storage regarding SSB data, in particular that of integer base data and query intermediate results.

Data Type	Unencoded	*AHEAD*		AN Bit-Packed			
		$\eta \in \{1,2,3\}$	$\eta = 4$	$\eta = 1$	$\eta = 2$	$\eta = 3$	$\eta = 4$
`tiny`	8	16	32	10	13	16	19
`short`	16	32	32	18	22	25	29
`int`	32	64	64	34	39	42	47

Table 6.12: Storage consumption per data type in bits.

Table 6.12 summarizes the storage consumption for the individual integer data types, in bits, used for the Star schema benchmark for the Unprotected case and the *AHEAD* implementation. We consider $\eta \in \{1, \ldots, 4\}$, because 4 bit flips were measured in the RowHammer experiments [125]. Our *AHEAD* prototype supports only very limited integer compression, where we only use the smallest possible native register width for integer data, as described in Section 6.1.1. Therefore, *AHEAD* uses twice as many bits than the Unprotected baseline. $\eta = 4$ is a special case, because the encoded data type `AN_tiny` is mapped to the 16 bit `uint16_t` implementation type. This would only allow $\eta \leq 3$, because only 8 additional bits could be used for coding, but for $\eta = 4$ we would 11 additional bits (cf. Table 3.6). To enable this, we mapped `AN_tiny` to the next larger implementation type `uint32_t` which requires $4\times$ as many bits than the unencoded data type.

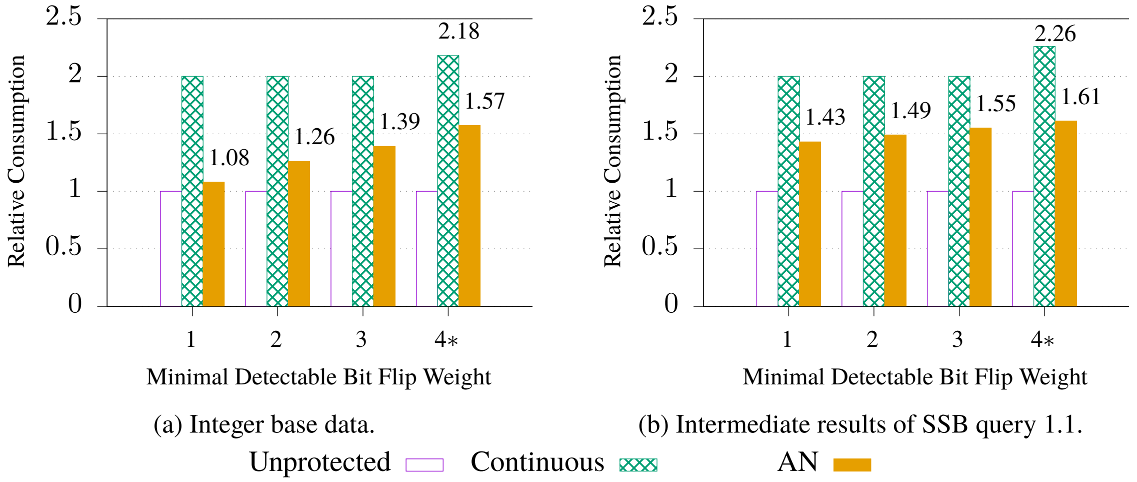
In Section 4.2.2 we discussed in detail how to combine AN coding and lightweight integer compression. Therefore, we consider in Table 6.12 also the case if we had enabled bit-packing for our *AHEAD* prototype. To calculate these numbers, for each data type we took the smallest respective A from Table 3.6 for each $\eta \in \{1, \ldots, 4\}$ and only added the actual bit width of each A. This is shown in the last four columns of Table 6.12. As we can see from the numbers, the storage overhead would be considerably lower per data type when storing data bit-packed.

We now consider the actual storage requirements for SSB. For the following, we assume scale factor 1 and those numbers can be scaled up for other SSB scale factors according to the cardinalities given in [131]. Figure 6.20 depicts the measured storage consumptions for base data (a) and SSB query 1.1 intermediate results (b). For the intermediate results, we added all sizes together. Furthermore, we now only consider the consumption for the Unprotected baseline and the Continuous Detection variant. The storage consumption is given relative to the unencoded case. Due to the *AHEAD* implementation, Continous storage consumption relative to the Unprotected case is currently 2 for $\eta \in \{1, 2, 3\}$. This is the same for both integer base data and query intermediate results. Since we had to increase the bit width of `AN_tiny` for $\eta = 4$, the storage overhead increases in that particular case from 100% to 118% for base data and to 126% for intermediate results. Figure 6.20 also shows the case if we had included bit-packing. In contrast to the fixed register widths, the storage consumption grows with increasing η. By that, instead of a constant $2\times$ storage consumption, it increases from $1.08\times$ gradually to $1.57\times$ for the base data, and from $1.43\times$ to $1.61\times$ for the intermediate data. The differences result from the different proportions of the integer types. As we could see in Figure 3.25, for smaller data types the relative coding overhead is much higher than for larger data types. For query 1.1 in particular, most of the intermediate result values are of the smaller data types. Therefore, the proportion of "expensive" coding overhead is much higher than for the base data.

6.3.4 Summary and Conclusions

In this section, we discussed the impact of AN coding on runtime and storage for SSB queries. The whole SSB suite consists of 13 queries in total, with a mixture of selections on one or two predicates, joins, aggregations, and groupings. We presented measurements for the query runtimes, then we discussed the impact of vectorization, and finally we examined the storage overhead.

For the individual query runtimes, we compared the two Unprotected and DMR baselines against the four error detection variants Early and Late Onetime Detection, and Continuous Detection and its Recoding variant. Compared to Unprotected, DMR always showed about the expected double runtime, for both scalar and vectorized execution. For scalar execution, Early Onetime was on average even slower than DMR, because all base columns had to be decoded first by the additional Δ



(a) Integer base data.

(b) Intermediate results of SSB query 1.1.

Figure 6.20: Memory overhead of the implemented AN coding and a projection for bit-packing. $*$: hardened data type `AN_tiny` mapped to 32-bit integer to enable large enough codewords.

operators. However, for vectorized execution, Early Onetime performed much faster with average runtime overheads of 33% for Skylake and 47% for Xeon Phi. In contrast, Late Onetime adds only a few Δ operators before the final aggregation or grouping operators. Additionally, these intermediate results are much smaller than the base columns, so that the runtime overheads of Late Onetime are only marginal, with 1% to 4% on average for both systems and both execution variants. Finally, the Continuous and Recoding Detection variants performed bit flip detection on each value. Continuous showed average runtime overheads of 14% and 17% on the Skylake system for Scalar and SSE execution, respectively. Recoding of each value only incurred one additional percent of runtime overhead. On the Xeon Phi system, both Continuous and Recoding had overheads of 11% in the scalar case and 12% for SSE. In total, Continuous and Recoding are not only in between Early and Late from the runtime overhead perspective, but they perform error detection on each and every column value.

While we before only considered runtimes relative to the Unprotected baseline, we then investigated the actual improvements on query runtimes through vectorization for SSB query 1.1. We could show that the proportion of the total query runtime for the vectorizable operators was reduced dramatically by 30% to 46% for both systems and all detection variants. Employing SSE could reduce the overall query runtimes by factors of 1.8× for Recoding up to 2.9× for Early on the Skylake system. For the Xeon Phi system, total runtimes of query 1.1 could be reduced by factors between 1.6× for Late, Continuous and Recoding, and 2.4× for Early. The Late Onetime variant could benefit most, because the Δ gained so much more through vectorization and because they made up a great amount of the total query runtime.

Finally, we discussed the storage overheads, again by the example of SSB query 1.1. Due to the *AHEAD* implementation of using full register widths, the storage overhead is always 2, except when enabling larger minimal detectable bit flip weights of e.g. $\eta = 4$. Then, the encoded `AN_tiny` data type had to be mapped to larger implementation types due to the larger parameter A. Then, the storage overhead increased to 118% for the base columns and to 126% for the intermediate results. Then, we considered the case when bit-packing was enabled for hardened data. This projection showed that the storage overhead was much smaller, starting at only 8% for the integer base data and at 43% for query 1.1 intermediates. It gradually grew for $\eta = 4$ to 57% and 61% for base data and intermediates, respectively. This showed that for AN coding the storage overhead can be scaled with the required detection capabilities.

6.4 ERROR DETECTING B-TREE

In Section 4.3 we introduced the Error Detecting B-Tree (EDB-Tree), derivatives of the ubiquitous B-Tree index structure. In the following, we will capture impact of the error detection primitives added in the various EDB-Tree variants. For the comparisons we use two baselines: first, the basic B-Tree and, second, its TMR variant. The impact on throughput will me measured for the two basic operations *lookup* of individual keys, and *insertion* of key-value-pairs.

6.4.1 Single Key Lookup

In the following, we will show the impact of error detection using our presented EDB-Tree variants on single key lookup throughput. For the measurements, we first inserted varying numbers of keys, starting at 1000 and exponentially increasing by a factor of 10 up to 10 000 000. We set the payload to the key itself and we query the tree with the same random number seed[5] which is used for the insertions. This allows to insert random numbers and to minimize the overhead during the lookup phase of re-generating the inserted keys. Our two baselines are the above presented B-Tree and its TMR variant. Since the latter performs all operations effectively thrice and must vote for a majority, a throughput of at most 1/3 can be expected. For all tree variants, we implement the key lookup inside a node using a binary search. The checksum variants of the EDB-Tree, however, first recompute the necessary checksums and compare against the stored ones and then run the binary search afterwards.

Figure 6.21 shows the lookup performance on system 1 (Skylake), where the upper two graphs show the absolute throughput numbers and the lower two graphs represent the throughput relative to the B-Tree baseline. Measurements for 32-bit keys and values are shown on the left (Figures 6.21a and 6.21c), while measurements for 64-bit keys and values are shown in the right graphs (Figures 6.21b and 6.21d). All index variants scale quite good, with lookup throughput degrading 35% on average with 10× increase of inserted keys. The decrease can be attributed to greater tree depth or higher fill levels in the nodes. In the former case, lookup traverses more nodes which leads to more random memory accesses. In the latter case, more keys are compared on each tree level and each node potentially has a higher fill level. From the relative throughput numbers we can derive that with increasingly more inserted keys, the overhead of the EDB-Tree variants is more and more reduced. The pure EDB-Tree and the parity bit variant (EDB-PB) get very close to the B-Tree baseline performance. In contrast, the checksum variants lead to much greater overheads. For both 32-bit and 64-bit, the lookup throughput increases from about 40% to 70%. As expected, the TMR B-Tree lookup performance is below one third of the original B-Tree, ranging from 30% to 32%.

Figure 6.22 shows the measurements for system 2 (Phi), with the same graph arrangement as before in Figure 6.22. While the absolute throughput numbers are around 4× lower than for the Skylake system, the most notable change is, that the checksum EDB-Tree variants are even slower than the TMR B-Tree for the smaller trees. However, for all EDB-Tree variants the overhead also decreases with larger tree sizes.

[5] We use an Xorshift random number generator, see e.g. `https://en.wikipedia.org/wiki/Xorshift`

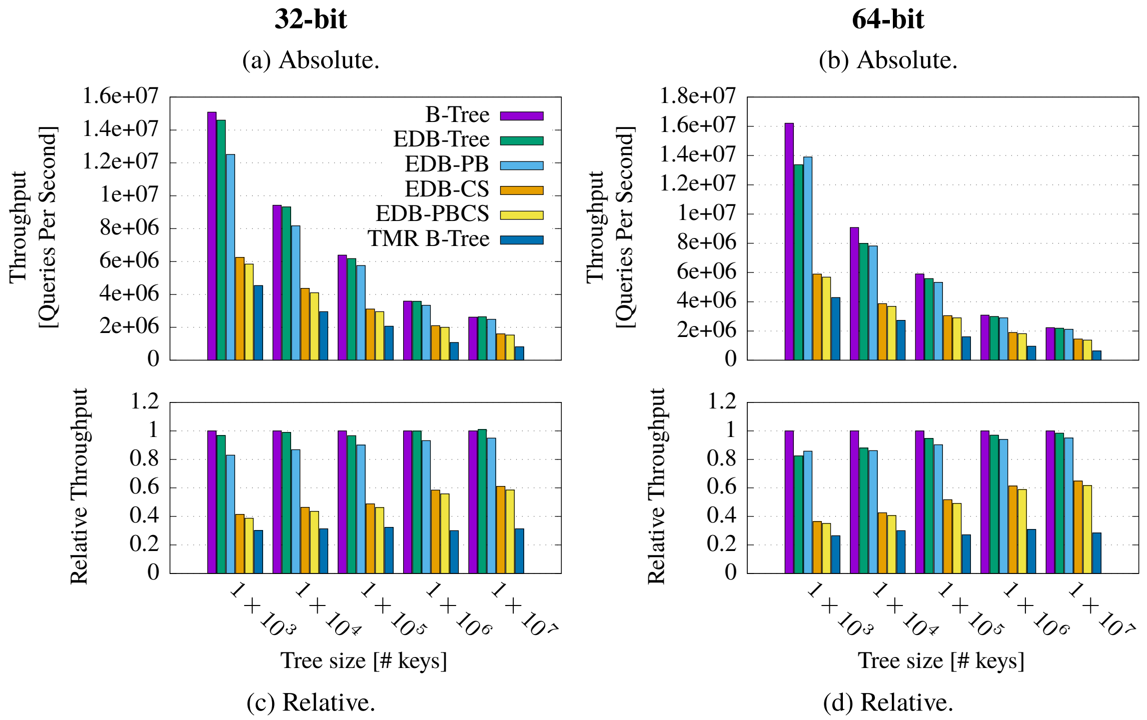


Figure 6.21: Lookup throughput for B-Tree and EDB-Tree variants on **system 1**.

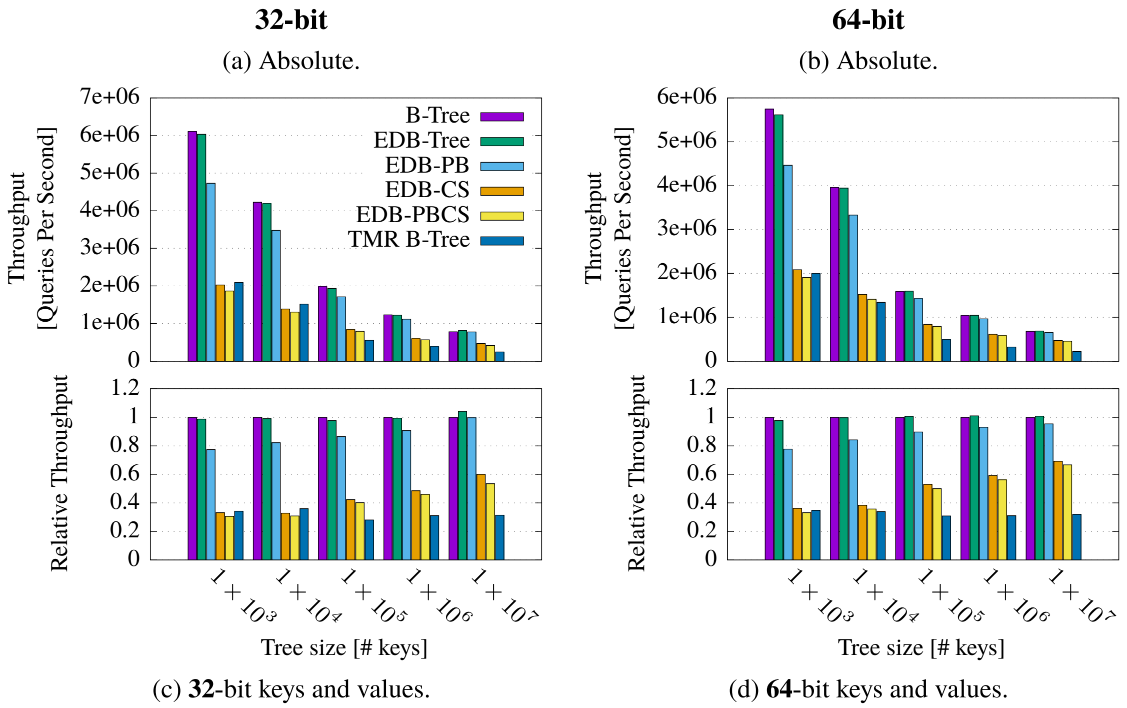
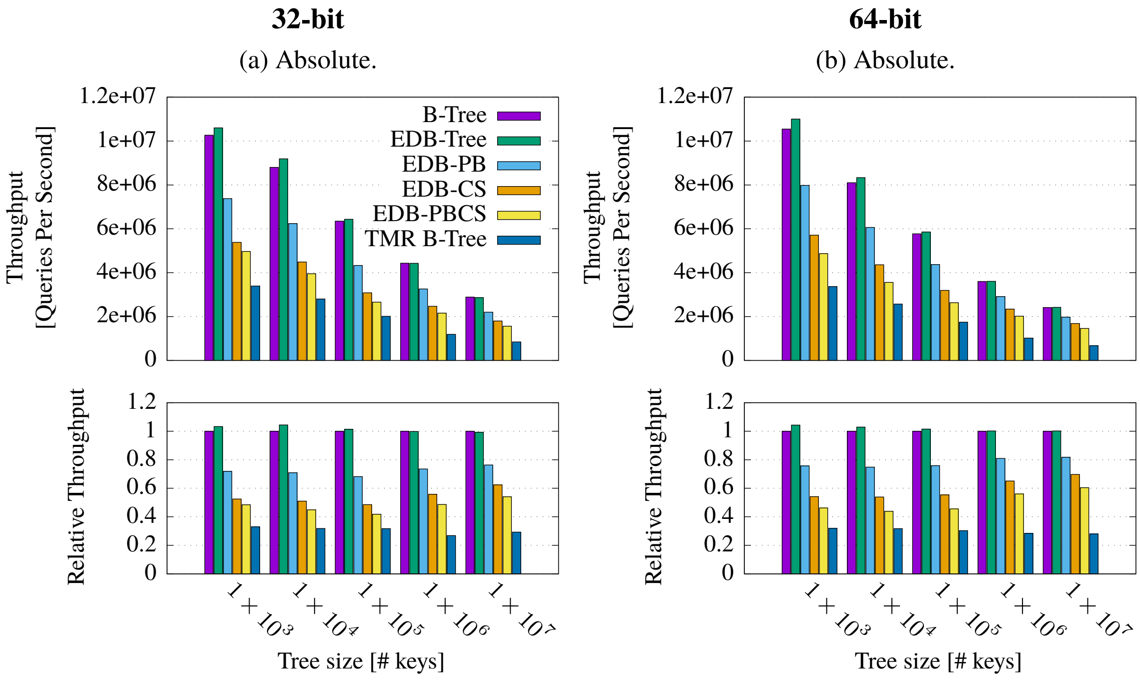


Figure 6.22: Lookup throughput for B-Tree and EDB-Tree variants on **system 2**.



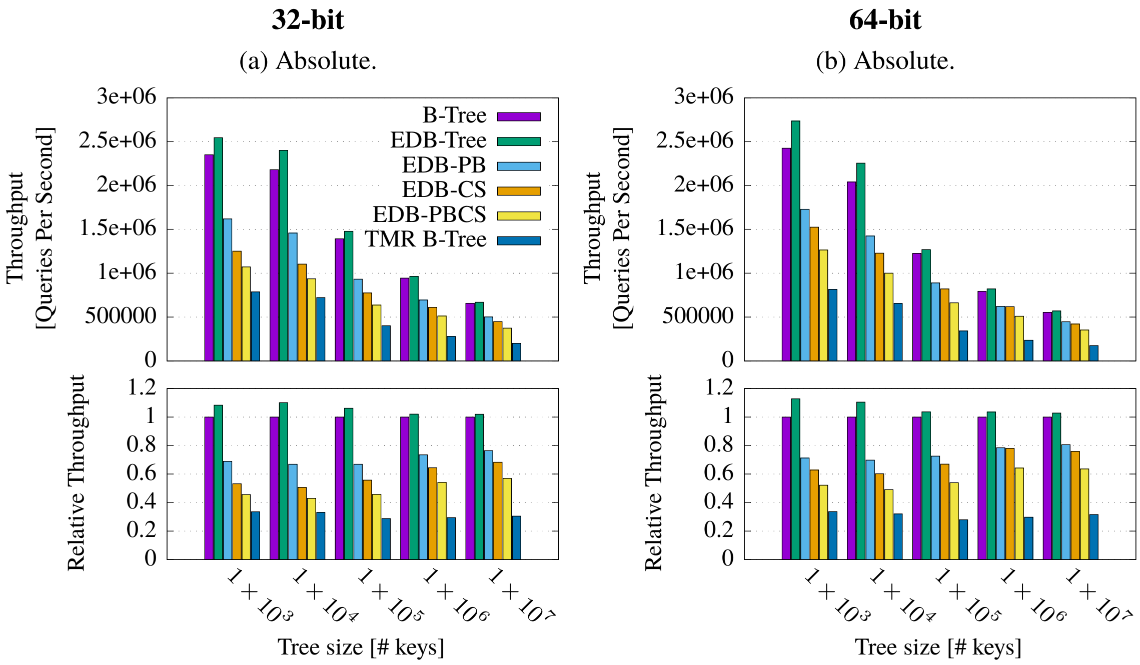
(a) Absolute.

(b) Absolute.

(c) Relative.

(d) Relative.

Figure 6.23: Insert throughput for B-Tree and EDB-Tree variants on **system 1**.



(a) Absolute.

(b) Absolute.

(c) **32**-bit keys and values.

(d) **64**-bit keys and values.

Figure 6.24: Insert throughput for B-Tree and EDB-Tree variants on **system 2**.

6.4.2 Key Value-Pair Insertion

Next, we measured runtimes for single key value-pair insertions. This operation basically includes the previous lookup to find the respective tree node where to insert the new key value-pair. Then, however, more work needs to be done to insert key and value in the sorted array inside the node. Furthermore, isnertions may include node splits where a new node must be allocated and keys, values, and pointers need to be moved to the new node. Additionally, for the EDB-Tree variants, child nodes' parent pointers need to be adjusted in such cases. By that, the single pair insertion should be lower on average than the pure lookup. This is confirmed by Figures 6.23 and 6.24, where the absolute throughput numbers are lower than those for the lookup (cf. Figures 6.21 and 6.22). Regarding the relative throughput, however, the picture is slightly different than for the lookup. Here, the EDB-Tree variant, which only employs pointer sanity checks, and the baseline are on par and the EDB-Tree is for very small trees even slightly faster. In contrast, the parity bit variant EDB-TreePB has slower relative throughput than before for all tree sizes. The checksums variants EDB-TreeCS and EDB-TreePBCS have slightly higher relative throughput for the smaller tree sizes and are virtually the same as before for the largest measured tree sizes. The TMR B-Tree throughput is again around one third of the B-Tree, as would be expected.

6.5 SUMMARY

In this chapter, we conducted an end-to-end evaluation of our error detection opportunities and the improved AN coding. We therefore built an in-memory prototype *AHEAD* for running all of the SSB queries and to also take detailed runtime and performance counter measurements of individual physical operators. We chose a greenfield approach, so that we could concentrate on those parts of the physical storage layer which we had to adapt for data and operator hardening. *AHEAD* provides a variety of physical data types, summarized in Table 6.1, as well as a several physical database operators, summarized in Table 6.2. We have shown that for comparing different physical data types, physical operators, and bit flip detection variants, including our detection opportunities, would require a huge amount of implementations, indicated in Table 6.2 and Figure 6.2. We reduced the complexity and effort by using modern C++ template constructs which allows to let the compiler generate the code of potentially more than 50 000 physical operator variants with just *49* code templates. This approach can be extended to more physical data types, sub-operations, and error control codes. We provided two concrete operator implementations in Listings 6.1 and 6.2 to illustrate the ease of data type adoption, where error detection primitives can be added just as it was previously outlined in Algorithms 5.3 and 5.4.

We provided detailed analysis of the runtimes of several individual physical operators, for both of our measurement systems from Section 3.3.1. The choice of operators included selection operators on one and two predicates, three different joins, as well as two grouping and two aggregation operators. For all of the scalar operator variants, the runtime overheads of Continuous Detection with and without recoding were in the range between $3.5\% \dots 14.9\%$ for the Skylake system and between $1.9\% \dots 24.7\%$ for the Xeon Phi system. Regarding vectorized performance, the coded operators' runtime overheads increased for the selection operators to $44\% \dots 53\%$ for Skylake and to $33\% \dots 35\%$ for Xeon Phi. For the last aggregation operator, the runtime overheads even increased to around 400% and 1000% for Skylake and Phi, respectively. This last operator, however, is purely memory read and compute intensive, where there are little to no stalling cycles where the CPU can hide the error detection instructions in between. The increase in runtime overhead means that the non-hardened operators scale better using vectorization than the AN coded ones. However, when there is more work to do in general in an operator, error detection has less impact on the operator runtime. This is also the case for the selection operators, where more data is written back to memory. Furthermore, we can see that recoding introduces virtually no additional overhead to error detection. Then, we compared the impact of vectorization on the Δ operator, since it has no unencoded counterpart. As we showed, it benefits exceptionally well, since the ratio between scalar and SSE runtimes were even greater than the ratio of data elements per vector register.

We conducted runtime measurements for all of the SSB queries. Next to the Unprotected baseline and our detection opportunities, we also considered a DMR variant which stores all data twice and executes each query twice, as well, followed by a voting on both query results. Throughout all tests and both systems, the DMR variants always showed the expected relative runtime of $\approx 2\times$. For Early Onetime detection, the scalar execution showed huge runtime overheads for the additional Δ operator for each column. This could be mitigated partly through vectorization, mostly because the Δ operator improved so much through vectorization. Late Onetime detection shows only very little runtime overheads, as was expected, since it performs error detection and decoding only before the last aggregation or grouping operator. The Continuous and Recoding variants showed small average overheads for both measurement systems and all queries of between 11% to 18%. This is an outstanding performance, because as we have seen in the very beginning in Section 2.5, most of the general-purpose techniques for bit flip detection showed much higher runtime or resource usage. This shows that our AN coding improvements, shown in Section 3.5, combined with the tight integration of error control codes into the physical storage layer, is a much more efficient way of detecting arbitrary bit flips in both business data and intermediate results. Using vectorization (SSE), the total query runtimes improved by $1.8\times\ldots 2.9\times$ on the Skylake system and by $1.6\times\ldots 2.4\times$ on the Xeon Phi system. The total runtime improvements are lower than the ratios of data elements per vector register, because not all operators are vectorized.

We also analyzed the storage consumption for base data and intermediate results for SSB query 1.1. As we showed, the limitations of the current *AHEAD* implementation lead to 100% storage overhead, which goes up when we increase the detection capabilities. Putting these limitations aside, assuming bit-packing, we showed that the storage overhead increases gradually with the detection capabilities. In particular, increasing $\eta = 1$ to $\eta = 4$, the storage overhead increased from 8% to 57% for base data and from 43% to 61% for intermediate data.

Finally, for the EDB-Tree variants we measured the throughput for single key lookup and single key value-pair insertion. The measurements revealed that the EDB-Tree, which only introduces pointer sanity checks, exhibits little to no throughput overhead. For lookups, on both systems, adding parity bit checks led to 20% less throughput for very small trees, but for very large trees the overhead was nullified. For insertions, the throughput reduction ranged from $\approx 30\%$ to $\approx 20\%$ on both systems. Introducing checksums even further degraded both lookup and insertion throughput, while the TMR B-Tree variant always showed the expected 1/3 throughput compared to the B-Tree baseline. Overall, for all EDB-Tree variants we saw that increasing tree sizes led to reduced throughput penalties compared to the baseline B-Tree. AN coding could also be used in EDB-Trees, but we leave the straight-forward investigation and performance evaluation to future work.

We dare to state that in this chapter we proved that our approach fulfills Requirement $\mathcal{R}2$ – *efficiency* – not only for coding operations, but also for query processing as a whole. Compared to the general purpose techqniques presented in Chapter 2, both runtime and storage overheads are greatly reduced. Furthermore, we showed that recoding of intermediates leads to virtually no additional runtime penalty, which underlines the fulfillment of Requirement $\mathcal{R}3$ – *adaptability*. This also proves that online bit flip detection during query processing is feasible and this fulfills Requirement $\mathcal{R}4$ – *availabilty*.

7

SUMMARY AND CONCLUSIONS

The key objective of database systems is to *reliably* manage data, while *high query throughput* and *low query latency* are core requirements. To satisfy these requirements, database systems constantly adapt to novel hardware features, such as vectorized computation using SIMD instructions. These hardware improvements are enabled in part by the constant decrease of transistor feature sizes. The downside of this miniaturization is proven by many studies: hardware produced with such tiny feature sizes becomes increasingly *unreliable*. Most if not all database research activity has been concentrating on the speed requirements (throughput and latency) and neglected the rising problem of transient hardware errors, called bit flips. It was shown that memory cells are much more error prone than logic gates. This is especially harmful for in-memory data management systems, where all business data is primarily stored and processed in fast main memory and where bit flips lead to corrupt query results in the form of e.g. false positives, false negatives, and wrong aggregates. While operating system and hardware could capture many of those errors, more and more field studies show that the error detection and correction capabilities are not sufficient any longer. An emerging and acknowledged trend in other research communities is to complement OS and hardware by detecting and correcting transient bit flips *also* in higher software layers. In this thesis, we investigated how transient hardware errors can be detected in an in-memory data management system, and try to smooth the way for a new database research domain.

At first, in Chapter 2 we defined Requirements $\mathcal{R}1$ to $\mathcal{R}5$ for a software-based solution of bit flip detection, respectively regarding effectiveness, efficiency, adaptability, availability, and separation of concerns. From these, *effectiveness* is the most important one, which means to provide the guarantee for a desired minimal detectable bit flip weight. The second most important requirement is *efficiency* and throughput the whole thesis, we provide throughput and speed measurements. Then, we compared contemporary techniques from several software domains and concluded that these do not completely satisfy our requirements. Since most if not all of the available software-based approaches utilize some sort of n-modular redundancy, we chose to use data encoding. Runtime measurements throughout this thesis verify that we chose extremely fast coding schemes.

We compared in Chapter 3 the following three code families regarding their effectiveness, efficiency, and adaptability: Hamming codes, XOR checksums, and arithmetic AN codes. As it turned out, software Hamming coding exhibits high probabilities of silent data corruption and very high coding operation runtimes, while the latter codes exhibit much better detection capabilities. However, AN coding also showed dreadful coding runtimes. To the best of our knowledge, in more than 60 years since the first mentioning of AN codes that we found, we are the first to use an elegant mathematical solution – the modular multiplicative inverse. This greatly simplifies error detection and decoding, while it also enables complete vectorization of the coding operations. This is especially important for main memory-centric DMSs and we measured that it can greatly improve coding runtimes by up to more than 18×[1]. Based on the different characteristics of the two code families, we concluded that both can be used at different places in DMSs and that both potentially satisfy the remaining Requirements $\mathcal{R}1$ to $\mathcal{R}4$. Both are effective regarding their error detection capabilities, as well as efficient regarding memory consumption and coding latencies. Finally, while XOR checksums are *adaptable* only to a limited degree, AN codes can be tailored to any *data bit widths* as well as desired *minimal detectable bit flip weights*.

Using the two code families, we introduced our *bit flip detecting storage* in Chapter 4, which is an adaptation of only the lowest physical storage layer. As a consequence, the conceptual and external layers can still provide the abstraction towards client applications, without being changed. We showed how the two coding schemes neatly integrate with many different aspects of that layer.

[1] The benchmarks suite we used for these measurements is open source available under `https://github.com/brics-db/coding_benchmark/`. Descriptions can be found at `https://brics-db.github.io/coding_benchmark/`

In-memory column stores store only integers in their base columns and these can be protected using AN coding. For variable-width data like strings, they employ dictionary coding to keep only integers in the base columns, while lightweight integer compression reduces the required memory footprint of base data. As we discussed, AN coding can be well combined with integer lightweight compression techniques. Strings can be protected using XOR checksums when higher speed is desired, or each character could be AN coded when better error detection is desired. Furthermore, we hardened the ubiquitous B-Tree index structure against bit flips. The important difference to column data is, that index structures have *structure* which can be exploited to detect bit flips. For instance, pointer sanity checks are one simple method, and XOR checksums can be used to protect keys and payload separately.

Based on the hardened storage layer, we introduced *bit flip detecting query processing* in Chapter 5, where we concentrated on the peculiarities of in-memory column stores. We discussed several *detection opportunities*, where inside a query bit flips can be deteted. We started with *Early* and *Late Onetime detection*, which introduce a new Δ operator doing error detection and decoding on base columns. While this requires comparatively little integrtation effort, bit flips are detected only in the base data or very late during a query and wrong join partners, wrong aggregates and more are still possible. *Continuous Detection* leverages the improved AN coding and does error detection in each and every operator, on each and every data unit. On the one hand, this requires more engineering effort than for Early and Late, because all physical operators must be adapted to include error detection primitives, but on the other hand this allows very fine-grained error detection. The hardening of intermediate results is another important topic we discussed. Using bitmaps to postpone materialization of intermediates, the compromise between speed, in favor of checksums, and detection capability, in favor of AN coding, must be made. Instead, full materialization can be used where selective stores should be favored instead of table lookup approaches, because for the latter, large lookup tables must be kept in unreliable main memory.

To prove that our approaches are indeed feasible in the sense that they can actually be employed in an online fashion during query execution, we presented an end-to-end evaluation in Chapter 6. Therefore, we implemented the open source[2] *AHEAD* prototype. We demonstrated that the engineering effort to integrate Continuous Detection can be mitigated elegantly through C++ template programming. In theory we could have instantiated $53\,141$ physical operators with merely *49* operator code templates. We discussed a few of these templates, as well as the additions for AN coding aware operators. We then discussed runtime measurements for individual operators based on SSB queries. One principal insight was that powerful systems such as the Skylake test system can nicely schedule the additional coding instructions. This can hide the coding overhead to a great degree, which results in less runtime overhead than for weaker systems such as the Xeon Phi test system. A further insight is that, the more complex the operators, the smaller the impact of additional coding operations. Also, for the individual operators, recoding had little to no impact on the measured runtimes. Then, we measured the total runtimes for all SSB queries on both test systems. The Continuous and Recoding variants showed small average overheads for both measurement systems and all queries of between 11% to 18%. These are excellent results, because as we have seen in Section 2.5, most of the general-purpose techniques for bit flip detection showed much higher runtime or resource overheads. This shows that our AN coding improvements, combined with the tight integration in the storage layer and operators, is a much more efficient way of detecting arbitrary bit flips in both business data and intermediate results. As we furthermore discussed, the storage overhead of the *AHEAD* prototype is currently 100% and above, depending on the desired minimal detectable bit flip weight. Using bit packing techniques, this overhead can be greatly reduced. In particular, for SSB query 1.1, the storage overhead ranges from 8% to

[2] The whole project `https://brics-db.github.io`

57% for base data and from 43% to 61% for intermediate data. Finally, for the EDB-Tree variants we also measured the throughput for single key lookup and single key value-pair insertion. The measurements revealed that the pointer sanity checks introduce little to no throughput overhead. For the more expensive variants parity bits and XOR checksums, the overheads were much greater for smaller trees. For all EDB-Tree variants we saw that increasing tree sizes led to reduced throughput penalties compared to the baseline B-Tree.

In total, our presented bit flip detecting storage and Continuous Detection approaches fulfill all our Requirements $\mathcal{R}1$ to $\mathcal{R}5$. Regarding effectiveness ($\mathcal{R}1$), we use AN coding, for which a parameter A can be chosen to cover e.g. desired minimal detectable bit flip weights, while XOR checksums can be used for large or arbitrary length blocks of data. Regarding efficiency ($\mathcal{R}2$), both code families are extremely fast software-based coding schemes, especially AN coding with our performance improvements. Furthermore, for both codes all operations can be vectorized using SIMD instruction sets. Regarding adaptability ($\mathcal{R}3$), AN coding allows to adapt the code strength at runtime, even during query processing for intermediate results. Recoding is extremely fast due to our use of the modular multiplicative inverse. Regarding availability ($\mathcal{R}4$), we use an online approach, which does error detection at runtime for each and every data element from base columns and intermediate results. Also, our hardened EDB-Tree does bit flip detection in an online fashion during each tree traversal. Regarding separation of concerns ($\mathcal{R}5$), we care only about that part which we know best – our managed data and data structures.

With this work, we hope to pave the way for future research, which may build upon, extend, or complement the approaches presented in this thesis. Most of the source code fragments used to obtain the results in this thesis are open source and can be found through the project page `https://brics-db.github.io` or directly on GitHub at `https://github.com/brics-db/`.

7.1 FUTURE WORK

We believe that data management system must take care of transient hardware errors like bit flips, as one part of the whole hardware / software stack. Our error code analysis and AN coding improvements, as well as the hardened storage and query processing layers are one major step towards bit flip detecting and correcting DMSs. In the following, we discuss further research directions which we believe are very promising and important.

Hardware Error Models

One important question, which we can not answer in this thesis, is that for concrete hardware error models. While we based our discussions on a fairly abstract error model, concrete ones may loosen or even tighten the requirements on the employed error codes. The ones we discussed in Section 3.2.4 are not derived from actual hardware and were merely used to demonstrate how the conditional probabilities of the error codes can be transformed into unconditional ones. Actual hardware error models may dictate that some error patterns are more likely than others, in contrast to those assumptions in Section 3.2.4. Furthermore, new optimality criteria could be expressed, which are different from those we presented in Section 3.2.3. For instance, if large burst errors were actually more likely than individual bit flips, then other golden As than in Table 3.6 might be required. To take this one step further, consider the problem of aging, which implies that the hardware error model changes over time. Here, the interesting question is, when the objective functions, which describe the optimality criterion, are provided. For instance, they could be provided at software design or compile time by the hardware vendors. This would be a *static* approach. In contrast, the hardware itself or some software monitoring service could measure the state of the hardware (or the system as a whole) and deliver characteristics to the DMS. This one, in turn, might then choose the code parameters from a different set of golden As, in the case of AN coding. This opens a whole new area of question, like how a running system's error model can be measured, when and how new coding parameters are chosen, or whether such models can be *forecasted*.

Coding Tradeoffs

Concerning the effectiveness and efficiency of coding schemes, both are often contradicting goals. The AN coding speed may even more be increased, e.g. through the use of so-called *accumulators* [13, 93, 154]. The idea is to accumulate multiple code words before doing the actual error detection. This works, due to the distributivity of the arithmetic code operations. Using an accumulator, less error detection operations must be executed. It would be crucial to know about the error propagation and how the SDC probabilities change when employing accumulators. Furthermore, the size of the accumulator is of importance: when too many values are added up, then an overflow may happen, which very likely results in a non-codeword. Using accumulators with a larger data width might require a different A and recoding. Another interesting topic to this regard is the use of coded operators when queries are just-in-time compiled [128]. Here, whole pipelines of operators are put together and it should be investigated whether and, if so, how often intermediates in this pipeline must be checked for bit flips. Furthermore, it would be interesting to see the query runtime overhead when compiling several operators together. Since more work is done per single data element, the coding overhead should be further reduced.

Bit Flip Detecting Storage

We are currently at the brink of new, emerging memory technologies with different performance characteristics than current DRAM technology. For instance, non-volatile main memory will reach the markets in the near future, where read and write latencies greatly differ. It would be very interesting to investigate the impact of such future main memory technologies on bit flip handling DMSs. Another important point is the adaptation of EDB-Trees with AN coding. Regarding the pointers, since not all 64 bits of pointers are used for addresses, we can use these spare bits for coding. However, there are two issues. First, since now up to 56 bits can be used for virtual addresses, there are only 8 bits remaining, which leaves only very little room for adapting to increasing bit flip weights. Second, since it is very hard to compute the SDC probabilities for such large data widths, we have not yet computed any golden or super As. Consequently, we need to reduce the data size of the pointers. Employing AN coding to protect the keys and payloads may be more trivial.

Bit Flip Detecting Processing

In this thesis, we only considered the actual query execution, but the whole query life cycle are composed of so many more parts, not to speak of a whole DMS. For instance, as soon as a query reaches the DMS (typically in form of a string), all state may have to be protected against bit flips.

For instance, it must be ensured, that the correct string is parsed and transformed to a valid internal query representation. Next, this internal representation must be protected as well, so that e.g. the correct operators are called. Finally, the results must be protected and reliably sent back to the original user.

Bit Flip Correction

This thesis only dealt with the *detection* of bit flips in data and data structures. Going from this, the *correction* of these errors is the next logical step. The error codes used in this thesis are detection-only codes, so that bit flips can not be corrected naively. Furthermore, as we have seen with the Hamming codes, using the same code for detection *and* correction leads to much worse bit flip detection capabilities. Therefore, we advocate to use error codes for error detection and different techniques for error correction. One possibility is to explore the use of inherent redundancy in DMSs. For instance, there is the database log for taking care of the contemporary failure classes, as well as materialized indexes and views. This means that the DMS stores its data probably several times at different locations, anyways. This could be used to not introduce even more redundancy, but to reproduce corrupted data from other, already existing locations. Another promising idea we thought about is to use *linear combinations* known from network coding and RAID systems. However, in contrast to these, instead of XOR-ing values together, we can exploit the properties of AN coding, namely add together two columns or parts of a single column. This would result in valid codewords, again, where recoding might be needed, and these could again be checked for bit flips. Furthermore, for the individual data structures of a DMS, different bit flip correction techniques might be necessary.

Hardened Processing in Memory

An interesting research direction is near-memory processing, or processing-in-memory (PIM) [5, 29]. Today, DRAM devices are already comprised of chip stacks with multiple layers of storage fabric. The basic idea of PIM is to integrate a logic layer into the bottom of the DRAM chip stack. This allows to offload simple tasks like data filtering to that logic layer and has several benefits. Firstly, this safes costly cycles in the CPU, so that the CPU can deal with more important or complicated tasks. Leveraging the improved AN coding, error detection could be done already at the device level, to reduce the load of the CPU. As we have seen in the *AHEAD* prototype description in Section 6.1.2, the huge diversity of physical operators due to the many possible data widths might pose an important challenge. Furthermore, the real benefits and tradeoffs of this combination are not yet clear. For instance, having checked the data at the device level, must the transferred values be checked at the CPU again? And if so, does this only benefit if whole operators can be offloaded to the DRAM device? These are only a few questions and many more seem to loom on the horizon.

A

APPENDIX

A.1 LIST OF GOLDEN AS

We provide the full list of golden As for all $k \in \{1, \ldots, 32\}$ and $|A| \in \{1, \ldots, 16\}$ in Tables A.1 to A.4. There, only bold numbers are prime and in contrast to previous suggestions [161], many super As are *not* prime. The minimum detection weight is denoted in parenthesis and all numbers marked with an asterisk (*) are grid approximated with $M = 1001$. What may be surprising is the fact that for a given width of A ($|A|$) and for increasing $k = |\mathbb{D}|$, the As are not strictly monotonically increasing. For instance, for $k \in \{3, 4, 5\}$ and $|A| = 7$, the super As are $89, 117, 115$ and for $|A| = 9$ and $k \in \{4, 5, 6$, the super As are $467, 443, 471$.

$\|A\|$				k				
	1	2	3	4	5	6	7	8
2	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)
3	**7** (2)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)
4	15 (3)	**13** (2)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)
5	**31** (4)	**29** (2)	**29** (2)	27 (2)	**29** (2)	**29** (2)	**29** (2)	**29** (2)
6	63 (5)	**53** (3)	45 (3)	45 (2)	45 (2)	45 (2)	**59** (2)	**59** (2)
7	**127** (6)	117 (3)	117 (3)	**89** (3)	117 (3)	115 (2)	115 (2)	115 (2)
8	255 (7)	213 (4)	**229** (3)	**229** (3)	**233** (3)	**233** (3)	217 (3)	**233** (3)
9	511 (8)	469 (4)	**467** (4)	**467** (3)	**443** (3)	471 (3)	471 (3)	**487** (3)
10	1023 (9)	**853** (5)	917 (4)	933 (4)	933 (4)	**809** (3)	933 (3)	**857** (3)
11	2047 (10)	**1877** (5)	1837 (5)	**1867** (4)	1909 (4)	1899 (4)	1803 (4)	1939 (4)
12	4095 (11)	3285 (6)	**3673** (5)	3737 (4)	3787 (4)	3813 (4)	3813 (4)	3813 (4)
13	**8191** (12)	6613 (6)	**7349** (6)	6777 (5)	7085 (5)	7837 (5)	7637 (4)	7463 (4)
14	16383 (13)	13141 (7)	13779 (6)	14937 (5)	15221 (5)	**14159** (5)	**13963** (5)	**13963** (5)
15	32767 (14)	26453 (7)	23733 (7)	31385 (6)	31373 (6)	31373 (5)	27247 (5)	27247 (5)
16	65535 (15)	52565 (8)	**56501** (7)	47729 (6)	59973 (6)	62739 (6)	55831 (6)	55831 (6)

Table A.1: Golden As for $k \in \{1, \ldots, 8\}$, $|A| \in \{1, \ldots, 16\}$ with minimal detection weight in parentheses, after $\Psi_\eta(k, h)$ – Equation (3.52). **Bold** numbers are prime.

$\|A\|$				k				
	9	10	11	12	13	14	15	16
2	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)
3	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)
4	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)
5	**29** (2)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)
6	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)
7	123 (2)	123 (2)	123 (2)	123 (2)	123 (2)	119 (2)	119 (2)	119 (2)
8	185 (3)	185 (3)	**233** (2)	215 (2)	215 (2)	**233** (2)	**233** (2)	**233** (2)
9	**487** (3)	451 (3)	451 (3)	**463** (3)	**463** (3)	**463** (3)	**463** (3)	**463** (3)
10	**857** (3)	**857** (3)	**857** (3)	**857** (3)	**947** (3)	**947** (3)	**947** (3)	**947** (3)
11	1939 (4)	1939 (3)	1939 (3)	1939 (3)	1939 (3)	1939 (3)	1939 (3)	1939 (3)
12	3621 (4)	**3739** (4)	**3739** (4)	3737 (4)	3349 (4)	3349 (3)	3349 (3)	3349 (3)
13	5729 (4)	6717 (4)	6717 (4)	6717 (4)	7413 (4)	6717 (4)	7785 (4)	7785 (4)
14	15717 (5)	15469 (4)	14139 (4)	14139 (4)	14139 (4)	14139 (4)	14139 (4)	14781 (4)
15	27425 (5)	27425 (5)	27425 (5)	29925 (5)	27825 (5)	**28619** (4)	**28183** (4)	**28183** (4)
16	55831 (6)	59965 (5)	**58901** (5)	62749 (5)	62749 (5)	63877 (5)	63877 (5)	63877 (5)

Table A.2: Golden As for $k \in \{9, \ldots, 16\}$, $|A| \in \{1, \ldots, 16\}$ with minimal detection weight in parentheses, after $\Psi_\eta(k, h)$ – Equation (3.52). **Bold** numbers are prime.

$\lvert A\rvert$	k							
	17	18	19	20	21	22	23	24
2	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)
3	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)
4	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)
5	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)
6	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)	**61** (2)
7	119 (2)	111 (2)	111 (2)	111 (2)	111 (2)	111 (2)	111 (2)	111 (2)
8	**233** (2)	**233** (2)	**233** (2)	**233** (2)	**233** (2)	237 (2)	237 (2)	237 (2)
9	393 (3)	393 (2)	393 (2)	393 (2)	423 (2)	423 (2)	423 (2)	423 (2)
10	**947** (3)	**947** (3)	**947** (3)	985 (3)	985 (3)	985 (3)	985 (3)	981 (3)
11	1909 (3)	1909 (3)	1939 (3)	1939 (3)	1909 (3)	1939 (3)	1939 (3)	1939 (3)
12	3349 (3)	3349 (3)	3349 (3)	3091 (3)	3091 (3)	3091 (3)	3091 (3)	3829 (3)
13	7785 (4)	7785 (4)	7985 (4)	7985 (4)	**6311** (3)	**6311** (3)	**6311** (3)	**6311** (3)
14	14781 (4)	15207 (4)	16089 (4)	16089 (4)	15507 (4)	15993 (4)	15993 (4)	15993 (4)
15	32343 (4)	32343 (4)	32343 (4)	32343 (4)	30987 (4)	30987 (4)	30987 (4)	29675 (4)
16	63859 (5)	63859 (5)	58659 (4)	58659 (4)	58659 (4)	58659 (4)	58659 (4)	64311 (4)

Table A.3: Golden As for $k \in \{17, \ldots, 24\}$, $|A| \in \{1, \ldots, 16\}$ with minimal detection weight in parentheses, after $\Psi_\eta(k, h)$ – Equation (3.52). **Bold** numbers are prime.

$\lvert A\rvert$	k							
	25	26	27	28	29	30	31	32
2	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)	**3** (1)
3	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)	**7** (1)
4	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)	**13** (1)
5	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)	**29** (1)
6	**61** (1)	**61** (1)	**61** (1)	**61** (1)	**61** (1)	**61** (1)	**61** (1)	**61** (1)
7	111 (2)	111 (2)	111 (2)	111 (2)	111 (2)	125 (2)	125 (2)	125 (2)
8	225 (2)	225 (2)	225 (2)	225 (2)	225 (2)	225 (2)	225 (2)	225 (2)
9	423 (2)	423 (2)	423 (2)	423 (2)	423 (2)	445 (2)	445 (2)	445 (2)
10	981 (3)	981 (3)	951 (3)	951 (3)	835 (3)	835 (3)	**881** (3)	**881** (3)
11	1939 (3)	1939 (3)	1939 (3)	1939 (3)	1939 (3)	1939 (3)	**2029** (3)	**2029** (3)*
12	3829 (3)	3829 (3)	3829 (3)	3829 (3)	3829 (3)	3829 (3)	3973 (3)*	3973 (3)*
13	**6311** (3)	**7841** (3)	**7841** (3)	**7841** (3)	**7841** (3)	**7841** (3)*	**7841** (3)*	**7841** (3)*
14	15685 (4)	15203 (4)	15203 (4)	15203 (3)	15203 (3)*	15203 (3)*	**15823** (3)*	16089 (3)*
15	29685 (4)	29685 (4)	29685 (4)	29685 (4)*	29685 (4)*	31693 (4)*	32211 (4)*	32417 (4)*
16	64311 (4)	64311 (4)	64311 (4)*	64311 (4)*	64311 (4)*	64311 (4)*	64665 (4)*	65117 (4)*

Table A.4: Golden As for $k \in \{25, \ldots, 32\}$, $|A| \in \{1, \ldots, 16\}$ with minimal detection weight in parentheses, after $\Psi_\eta(k, h)$ – Equation (3.52). **Bold** numbers are prime. Those marked with an asterisk (*) are obtained through grid approximation.

A.2 MORE ON HAMMING CODING

A.2.1 Code examples

Table A.5 lists some code examples and the appropriate Hamming weights, which exemplifies shortening of the Extended Hamming codes. In particular, the standard $(7,4)$ Hamming code is shown in Table A.5a in combination with its $(8,4)$ extended counterpart with the additional overall parity bit (p_0^*) and the other code word weights $(w_{\mathrm{H}}^{(8,4)})$. For comparison, the shortened extended codes $(7,3)$ and $(6,2)$ are shown in Tables A.5b and A.5c, respectively.

position:	7 6 5 3	4 2 1 0*	$w_{\mathrm{H}}^{(7,4)}$	$w_{\mathrm{H}}^{(8,4)}$
element:	$d_4d_3d_2d_1$	$p_3p_2p_1p_0^*$		
	0 0 0 0	0 0 0 0	0	0
	0 0 0 1	0 1 1 1	3	4
	0 0 1 0	1 0 1 1	3	4
	0 0 1 1	1 1 0 0	4	4
	0 1 0 0	1 1 0 1	3	4
	0 1 0 1	1 0 1 1	4	4
	0 1 1 0	0 1 1 0	4	4
	0 1 1 1	0 0 0 1	3	4
	1 0 0 0	1 1 1 0	4	4
	1 0 0 1	1 0 0 1	3	4
	1 0 1 0	0 1 0 1	3	4
	1 0 1 1	0 0 1 0	4	4
	1 1 0 0	0 0 1 1	3	4
	1 1 0 1	0 1 0 0	4	4
	1 1 1 0	1 0 0 0	4	4
	1 1 1 1	1 1 1 1	7	8

(a) Code words of (7,4) and (8,4)-Extended Hamming codes.

position:	6 5 3	4 2 1 0*
element:	$d_3d_2d_1$	$p_3p_2p_1p_0^*$
	0 0 0	0 0 0 0
	0 0 1	0 1 1 1
	0 1 0	1 0 1 1
	0 1 1	1 1 0 0
	1 0 0	1 1 0 1
	1 0 1	1 0 1 1
	1 1 0	0 1 1 0
	1 1 1	0 0 0 1

(b) Code words of (7,3) shortened Extended Hamming codes.

position:	5 3	4 2 1 0*
element:	d_2d_1	$p_3p_2p_1p_0^*$
	0 0	0 0 0 0
	0 1	0 1 1 1
	1 0	1 0 1 1
	1 1	1 1 0 0

(c) Code words of (6,2) shortened Extended Hamming Code.

Table A.5: Code words and weights of the (7,4) Hamming code, its (8,4) Extended variant ((a)), and the (7,3) and (6,2) shortened variants of the Extended Hamming code ((b) and (c), respectively). For the latter two, all code words have weight 4 – this property is not lost by shortening.

A.2.2 Vectorization

Now, we provide more details on the implementation of the techniques presented in Section 3.3.2. First, we will give more details on the population count implementation and afterwards discuss the actual hardening operation.

Population Count

In Section 3.3.2 we gave a brief introduction of the idea of vectorized population counting, which is the most crucial, since mostly used, instruction for computing the individual parity bits of the Hamming codes. There, we discussed two different SIMD ISAs, namely SSE4.2 and AVX2 and here we also present detailed numbers for both. Figure A.1 exercises the example given in Figure 3.18a, where 4 bits are set, with all intermediate steps and concrete values, while the data value is actually the same.

Operation	Value (binary)
`data`	`0110 1100`
`temp = data » 1`	`0011 0110`
`pattern1`	`0101 0101`
`temp = temp & pattern1`	`0001 0100`
`-temp`	`1110 1100`
`temp = data + (-temp)`	`0101 1000`
`pattern2`	`0011 0011`
`temp2 = temp & pattern2`	`0001 0000`
`temp = temp » 2`	`0001 0110`
`temp = temp & pattern2`	`0001 0010`
`temp = temp + temp2`	`0010 0010`
`pattern3`	`0001 0001`
`temp = temp * pattern3`	`0100 0010`
`result = temp » 4`	`0000 0100`

Figure A.1: Reduced example for 8-bit population count after Figure 3.18a. Bits are again grouped into 4 bits for better readability.

Listing A.1 gives a concrete vectorized implementation for 8-bit (single byte) values using SSE4.2. The same can be translated to AVX2 / AVX512 / . . . instructions. On some hardware platforms, Intel® Knights Mill in particular, a vectorized population count may be available through the `vpopcnt` instruction, which can make the binary adder partly obsolete. For Knights Mill the instruction is supposed to only support vectors of 16 32-bit or 8 64-bit data words, so that for smaller data words we have to fall back to the binary adder variant.

Function `_mm_popcount_epi8` uses several optimizations to reduce the total number of instructions (from [182]). In the first iteration the shifted, masked temporary (named `shifted`) is subtracted from the original data, which avoids ANDing the original data with the first pattern (Lines 7 and 8). The other two iteration adhere to the original algorithm (Lines 9 to 12). For single bytes, the multiplication bears no simplification, because the multiplication and the required shifting are more expensive than the addition and the masking.

Population count implementations for SSE4.2 on vectors of 16-bit and 32-bit data are shown in Listing A.2. There, a `uint64_t` (`uint32_t`) is a 64-bit (32-bit) wide unsigned integer that is returned from the function (Lines 1 and 11). We use a single temporary (Lines 2 and 12) and a single pattern for the multiplication (Lines 3 and 13), whereas the highest order bytes contain the actual population count (cf. Figure 3.18e). For up to 128-bit data elements we can use single bytes to store the population counts, so after the actual counting we can compact the result to save memory space. For that, we use a single shuffle mask (Lines 4 and 14). We first obtain the individual byte counts (Lines 5 and 15), then multiply to add them up (Lines 6 and 16) and shuffle

```
__m128i _mm_popcount_epi8(__m128i data) {
  __m128i temp;
  __m128i shifted;
  __m128i pattern1 = _mm_set1_epi8(0x55);
  __m128i pattern2 = _mm_set1_epi8(0x33);
  __m128i pattern3 = _mm_set1_epi8(0x0F);
  shifted = _mm_and_si128(_mm_srli_epi16(data, 1), pattern1);
  temp = _mm_sub_epi8(data, shifted);
  shifted = _mm_and_si128(_mm_srli_epi16(temp, 2), pattern2);
  temp = _mm_add_epi8(_mm_and_si128(temp, pattern2), shifted);
  shifted = _mm_and_si128(_mm_srli_epi16(temp, 4), pattern3);
  temp = _mm_add_epi8(_mm_and_si128(temp, pattern3), shifted);
  return temp;
}
```

Listing A.1: Vectorized SSE4.2 population count for 8-bit data, after [182, Section 5.1, p. 82]. The function name corresponds to the typical SSE4.2 naming pattern.

```
 1 uint64_t _mm_popcount_epi16(__m128i data) {
 2   __m128i temp;
 3   __m128i pattern = _mm_set1_epi16(0x0101);
 4   __m128i shuffle = _mm_set_epi64x(0xFFFFFFFFFFFFFFFF, 0x0F0D0B0907050301);
 5   temp = _mm_popcount_epi8(data);
 6   temp = _mm_mullo_epi16(temp, pattern);
 7   temp = _mm_shuffle_epi8(temp, shuffle);
 8   return _mm_extract_epi64(temp, 0);
 9 }

11 uint32_t _mm_popcount_epi32(__m128i data) {
12   __m128i temp;
13   __m128i pattern = _mm_set1_epi16(0x0101);
14   __m128i shuffle = _mm_set_epi32(0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0x0F0B0703);
15   temp = _mm_popcount_epi8(data);
16   temp = _mm_mullo_epi32(temp, pattern);
17   temp = _mm_shuffle_epi8(temp, shuffle);
18   return _mm_extract_epi32(temp, 0);
19 }
```

Listing A.2: Vectorized SSE4.2 population count for 16-bit and 32-bit data, after [182, Section 5.1, p. 82]. The function names correspond to the typical SSE4.2 naming pattern. Variants for AVX2 / AVX512 / ... are constructed likewise.

them for compaction (Lines 7 and 17). Since there fit 8 16-bit integers and 4 32-bit integers into a single 128-bit vector, the functions return accordingly large results (64-bit and 32-bit integer, respectively).

Hardening

Listing A.3 shows an SSE4.2 implementation for parallel Hamming coding on 8 16-bit data words. The code words are organized so that first the 128 bits of data are stored consecutively followed by the overall 64 code bits, also consecutively. We store 64 code bits since we store each data unit's code bits in its own byte. All Hamming code bits are computed in parallel for all 8 data units of the vector. The function naming again follows the SSE conventions (Line 1). First, the Hamming bit set patterns are defined in Lines 2 to 6 (cf. Figure 3.3). Lines 8 to 10 initialize the hamming code bits and temporaries to zero. Then, we consecutively mask each of the data patterns (Lines 10, 12, 15, 18 and 21) and store the corresponding code bit, i.e. we create even parity by selecting the LSB from the population count of each subcount (Lines 11, 14, 17, 20 and 23). Meanwhile,

```
uint64_t _mm_hamming_epi16(__m128i data) {
  __m128i pattern1 = _mm_set1_epi16(0xAD5B);
  __m128i pattern2 = _mm_set1_epi16(0x366D);
  __m128i pattern3 = _mm_set1_epi16(0xC78E);
  __m128i pattern4 = _mm_set1_epi16(0x07F0);
  __m128i pattern5 = _mm_set1_epi16(0xF800);

  uint64_t hamming = 0;
  uint64_t tmp1(0), tmp2(0);
  tmp2 = _mm_popcount_epi16(_mm_and_si128(data, pattern1)) & 0x0101010101010101;
  hamming |= tmp2 << 1;
  tmp1 = _mm_popcount_epi16(_mm_and_si128(data, pattern2)) & 0x0101010101010101;
  tmp2 ^= tmp1;
  hamming |= tmp1 << 2;
  tmp1 = _mm_popcount_epi16(_mm_and_si128(data, pattern3)) & 0x0101010101010101;
  tmp2 ^= tmp1;
  hamming |= tmp1 << 3;
  tmp1 = _mm_popcount_epi16(_mm_and_si128(data, pattern4)) & 0x0101010101010101;
  tmp2 ^= tmp1;
  hamming |= tmp1 << 4;
  tmp1 = _mm_popcount_epi16(_mm_and_si128(data, pattern5)) & 0x0101010101010101;
  tmp2 ^= tmp1;
  hamming |= tmp1 << 5;
  hamming |= (_mm_popcount_epi16(data) + tmp2) & 0x0101010101010101;
  return hamming;
}
```

Listing A.3: Vectorized SSE4.2 Hamming code computation for 16-bit data. Each byte in the result stores the hamming code bits of the input vector's appropriate 16-bit element

SIMD ISA	$\lvert\mathbb{V}\rvert$	8			16		
		data units per vector	code bits per data	code bits per vector[a]	data units per vector	code bits per data	code bits per vector[a]
SSE4.2	128	16	5	128	8	6	64
AVX2	256	32	5	256	16	6	128
AVX512	512	64	5	512	32	6	256

SIMD ISA	$\lvert\mathbb{V}\rvert$	32			64		
		data units per vector	code bits per data	code bits per vector[a]	data units per vector	code bits per data	code bits per vector[a]
SSE4.2	128	4	7	32	2	8	16
AVX2	256	8	7	64	4	8	32
AVX512	512	16	7	128	8	8	64

Table A.6: Total Hamming code bits per vector size and data element type. [a]: Total code bits aligned to full byte boundaries.

we use the second temporary to generate the additional overall parity (across all data and code bits). We XOR the code bits since the popcounts returned by `_mm_popcount_epi16` are packed into a single 64-bit data unit and XOR does parallel parity computation (Lines 13, 16, 19 and 22) – we could use addition and masking but we are interested in the parity and thus XOR suffices. Finally, we store the overall parity bit by XORing against the overall data popcount (Line 24). The Hamming computation can be adopted the same way for the other SIMD instruction sets (AVX2, AVX512, . . .) and other data widths. Table A.6 summarizes the amount of *actual* code bits for the Extended Hamming code with SECDED capability. For data element types up to 64-bit a single byte suffices for storing the extended Hamming code's code bits.

BIBLIOGRAPHY

[1] D. J. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems”, in *SIGMOD*, 2006, pp. 671–682.

[2] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, and T. Walter, “The beckman report on database research”, *Commun. ACM*, vol. 59, no. 2, pp. 92–99, 2016.

[3] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden, “The design and implementation of modern column-oriented database systems”, *Foundations and Trends in Databases*, vol. 5, no. 3, pp. 197–280, 2013.

[4] D. Abadi, S. Madden, and M. Ferreira, “Integrating compression and execution in column-oriented database systems”, in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’06, New York, NY, USA: ACM, 2006, pp. 671–682, ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142548.

[5] J. Ahn, S. Yoo, O. Mutlu, and K. Choi, “Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture”, in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ser. ISCA ’15, Portland, Oregon: ACM, 2015, pp. 336–348, ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750385. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750385.

[6] A. Ailamaki, D. J. DeWitt, and M. D. Hill, “Data page layouts for relational databases on deep memory hierarchies”, *The VLDB Journal*, vol. 11, no. 3, pp. 198–215, 2002.

[7] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, “Weaving relations for cache performance.”, vol. 1, pp. 169–180, 2001.

[8] AnandTech. (Jan. 11, 2018). Memblaze launches pblaze5 ssds: Enterprise 3d tlc, up to 6 gb/s, 1m iops, 11 tb, [Online]. Available: http://www.anandtech.com/show/11572.

[9] Arm Ltd. (Jul. 27, 2017). Arm NEON Technology, [Online]. Available: https://developer.arm.com/technologies/neon.

[10] T. Austin, “Diva: A reliable substrate for deep submicron microarchitecture design”, in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, 1999, pp. 196–207. DOI: 10.1109/MICRO.1999.809458.

[11] J. Autran, D. Munteanu, P. Roche, G. Gasiot, S. Martinie, S. Uznanski, S. Sauze, S. Semikh, E. Yakushev, S. Rozov, P. Loaiza, G. Warot, and M. Zampaolo, “Soft-errors induced by terrestrial neutrons and natural alpha-particle emitters in advanced memory circuits at ground level”, *Microelectronics Reliability*, vol. 50, no. 9–11, pp. 1822–1831, 2010, 21st European Symposium on the Reliability of Electron Devices, Failure Physics and Analysis, ISSN: 0026-2714. DOI: http://dx.doi.org/10.1016/j.microrel.2010.07.033.

[12] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, “Basic concepts and taxonomy of dependable and secure computing”, *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan. 2004, ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2.

[13] A. Avižienis, “Arithmetic error codes: Cost and effectiveness studies for application in digital system design”, *IEEE Trans. Computers*, vol. 20, no. 11, pp. 1322–1331, 1971.

[14] M. B.V. (Mar. 2, 2018). Monetdb goes headless, [Online]. Available: https://www.monetdb.org/blog/monetdb-goes-headless.

[15] D. Barbará, R. Goel, and S. Jajodia, "Using checksums to detect data corruption", English, in *Advances in Database Technology — EDBT 2000*, ser. Lecture Notes in Computer Science, C. Zaniolo, P. Lockemann, M. Scholl, and T. Grust, Eds., vol. 1777, Springer Berlin Heidelberg, 2000, pp. 136–149, ISBN: 978-3-540-67227-2. DOI: 10.1007/3-540-46439-5_9.

[16] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices", in *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, ser. SIGFIDET '70, New York, NY, USA: ACM, 1970, pp. 107–141. DOI: 10.1145/1734663.1734671.

[17] R. Bayer and K. Unterauer, "Prefix b-trees", *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 11–26, Mar. 1977, ISSN: 0362-5915. DOI: 10.1145/320521.320530.

[18] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores", in *SIGMOD*, 2009, pp. 283–296.

[19] P. Bohannon, R. Rastogi, S. Seshadri, A. Silberschatz, and S. Sudarshan, "Detection and recovery techniques for database corruption", *Knowledge and Data Engineering, IEEE Transactions on*, vol. 15, no. 5, pp. 1120–1136, Sep. 2003, ISSN: 1041-4347. DOI: 10.1109/TKDE.2003.1232268.

[20] M. Böhm, W. Lehner, and C. Fetzer, "Resiliency-aware data management", *PVLDB*, vol. 4, no. 12, pp. 1462–1465, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p1462-boehm.pdf.

[21] P. A. Boncz and M. L. Kersten, "MIL primitives for querying a fragmented world", *The VLDB Journal*, vol. 8, no. 2, pp. 101–119, Oct. 1999, ISSN: 1066-8888. DOI: 10.1007/s007780050076.

[22] P. A. Boncz, M. L. Kersten, and S. Manegold, "Breaking the memory wall in monetdb", *Commun. ACM*, vol. 51, no. 12, pp. 77–85, 2008.

[23] P. A. Boncz, M. Zukowski, and N. Nes, "Monetdb/x100: Hyper-pipelining query execution.", in *CIDR*, vol. 5, 2005, pp. 225–237.

[24] P. A. Boncz, "Monet; a next-generation dbms kernel for query-intensive applications", PhD thesis, University of Amsterdam, 2002.

[25] M. Borda, *Fundamentals in information theory and coding*. Berlin; Heidelberg: Springer, 2011, ISBN: 9783642203473. [Online]. Available: http://slubdd.de/katalog?TN_libero_mab215497729.

[26] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation", *Micro, IEEE*, vol. 25, no. 6, pp. 10–16, Nov. 2005, ISSN: 0272-1732. DOI: 10.1109/MM.2005.110.

[27] S. Y. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation", *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[28] S. Borkar and A. A. Chien, "The future of microprocessors", *Commun. ACM*, vol. 54, no. 5, pp. 67–77, 2011.

[29] A. Boroumand, S. Ghose, M. Patel, H. Hassan, B. Lucia, K. Hsieh, K. T. Malladi, H. Zheng, and O. Mutlu, "Lazypim: An efficient cache coherence mechanism for processing-in-memory", *IEEE Computer Architecture Letters*, vol. 16, no. 1, pp. 46–50, Jan. 2017, ISSN: 1556-6056. DOI: 10.1109/LCA.2016.2577557.

[30] L. Borucki, G. Schindlbeck, and C. Slayman, "Comparison of accelerated dram soft error rates measured at component and system level", in *Reliability Physics Symposium, 2008. IRPS 2008. IEEE International*, Apr. 2008, pp. 482–487. doi: 10.1109/RELPHY.2008.4558933.

[31] S. Breß, H. Funke, and J. Teubner, "Robust query processing in co-processor-accelerated databases", in *SIGMOD*, 2016, pp. 1891–1906.

[32] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations", *IRE Transactions on Electronic Computers*, vol. EC-9, no. 3, pp. 333–337, Sep. 1960, issn: 0367-9950. doi: 10.1109/TEC.1960.5219855.

[33] C.-Y. Chan and Y. E. Ioannidis, "An efficient bitmap encoding scheme for selection queries", *SIGMOD Rec.*, vol. 28, no. 2, pp. 215–226, Jun. 1999, issn: 0163-5808. doi: 10.1145/304181.304201.

[34] Z. Chen, J. Gehrke, and F. Korn, "Query optimization in compressed database systems", in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '01, New York, NY, USA: ACM, 2001, pp. 271–282, isbn: 1-58113-332-4. doi: 10.1145/375663.375692.

[35] D. Comer, "Ubiquitous b-tree", *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, Jun. 1979, issn: 0360-0300. doi: 10.1145/356770.356776.

[36] G. P. Copeland and S. N. Khoshafian, "A decomposition storage model", in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA: ACM, 1985, pp. 268–279, isbn: 0-89791-160-1. doi: 10.1145/318898.318923.

[37] P. Damme, D. Habich, J. Hildebrandt, and W. Lehner, "Lightweight data compression algorithms: An experimental survey (experiments and analyses)", in *EDBT*, 2017, pp. 72–83.

[38] F. David and R. Campbell, "Building a self-healing operating system", in *Dependable, Autonomic and Secure Computing, 2007. DASC 2007. Third IEEE International Symposium on*, Sep. 2007, pp. 3–10. doi: 10.1109/DASC.2007.22.

[39] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems", in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84, New York, NY, USA: ACM, 1984, pp. 1–8, isbn: 0-89791-128-8. doi: 10.1145/602259.602261.

[40] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: Sql server's memory-optimized oltp engine", in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, New York, New York, USA: ACM, 2013, pp. 1243–1254, isbn: 978-1-4503-2037-5. doi: 10.1145/2463676.2463710.

[41] J. M. Diamond, "Checking codes for digital computers", *Proceedings of the IRE*, vol. 43, no. 4, pp. 487–488, Apr. 1955, This letter reports the results of a study of checking codes made at the Moore School, University of Pennsylvania, in 1950-1951, on contract with the Burroughs Adding Machine Co., by Morris Plotkin and Joseph M. Diamond., issn: 0096-8390. doi: 10.1109/JRPROC.1955.277858.

[42] J. Do, Y. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges", in *SIGMOD*, 2013, pp. 1221–1230.

[43] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading", in *EMSOFT*, 2012, pp. 83–92.

[44] S. Electronics. (May 2018). Samsung starts mass producing industry's first 128-gigabyte ddr4 modules for enterprise servers, [Online]. Available: `https://news.samsung.com/global/samsung-starts-mass-producing-industrys-first-128-gigabyte-ddr4-modules-for-enterprise-servers`.

[45] P. Elias, "Coding for noisy channels", vol. 3, pp. 37–46, 4.

[46] M. Engel and B. Döbel, "The reliable computing base-a paradigm for software-based reliability.", in *GI-Jahrestagung*, 2012, pp. 480–493.

[47] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications", *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2012, ISSN: 0163-5808. DOI: `10.1145/2094114.2094126`.

[48] Z. Feng, E. Lo, B. Kao, and W. Xu, "Byteslice: Pushing the envelop of main memory data processing with a new storage layout", in *SIGMOD*, 2015, pp. 31–46.

[49] C. Fetzer, "Building critical applications using microservices", *IEEE Security & Privacy*, vol. 14, no. 6, pp. 86–89, 2016.

[50] A. Fog. (Feb. 2, 2018). Instruction tables, Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus, [Online]. Available: `Instruction%20tables`.

[51] P. Forin, "Vital coded microprocessor: Principles and application for various transit systems", *IFAC-GCCT*, pp. 79–84, Sep. 1989.

[52] Free Software Foundation. (Nov. 2016). The gnu multiple precision arithmetic library, [Online]. Available: `https://gmplib.org/`.

[53] R. A. Frohwerk, "Signature analysis: A new digital field service method", *Hewlett-Packard Journal*, vol. 28, no. 9, pp. 2–8, 1977.

[54] T. Fujiwara, T. Kasami, A. Kitai, and S. Lin, "On the undetected error probability for shortened hamming codes", *IEEE Transactions on Communications*, vol. 33, no. 6, pp. 570–574, 1985.

[55] M. Ghadhab, M. Kuntz, D. Kuvaiskii, and C. Fetzer, "A controller safety concept based on software-implemented fault tolerance for fail-operational automotive applications", in *Formal Techniques for Safety-Critical Systems*, C. Artho and P. C. Ölveczky, Eds., Springer International Publishing, 2016, pp. 189–205, ISBN: 978-3-319-29510-7.

[56] B. Gladman, W. Hart, J. Moxham, and et al. (Nov. 2016). MPIR: Multiple Precision Integers and Rationals, [Online]. Available: `http://mpir.org/`.

[57] M. J. E. Golay, "Notes on digital coding", *Proceedings of the Institute of Radio Engineers*, vol. 37, no. 6, pp. 657–657, 1949.

[58] J. Goldstein, R. Ramakrishnan, and U. Shaft, "Compressing relations and indexes", in *Proceedings of the 14th International Conference on Data Engineering*, Feb. 1998, pp. 370–379. DOI: `10.1109/ICDE.1998.655800`.

[59] S. Gordon, "Database integrity: Security, reliability, and performance considerations", *Indiana University South Bend, South Bend, IN*, vol. 12,

[60] G. Graefe and L. D. Shapiro, "Data compression and database performance", in *[Proceedings] 1991 Symposium on Applied Computing*, Apr. 1991, pp. 22–27. DOI: `10.1109/SOAC.1991.143840`.

[61] G. Graefe, "Volcano–an extensible and parallel query evaluation system", *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 1, pp. 120–135, 1994.

[62] G. Graefe and H. Kuno, "Definition, detection, and recovery of single-page failures, a fourth class of database failures", *Proceedings of the VLDB Endowment*, vol. 5, no. 7, 2012.

[63] G. Graefe, H. Kuno, and B. Seeger, "Self-diagnosing and self-healing indexes", in *Proceedings of the Fifth International Workshop on Testing Database Systems*, ser. DBTest '12, New York, NY, USA: ACM, 2012, 8:1–8:8, ISBN: 978-1-4503-1429-9. DOI: 10.1145/2304510.2304521.

[64] G. Graefe and R. Stonecipher, "Efficient verification of b-tree integrity", in *BTW*, 2009, pp. 27–46.

[65] G. Guzun and G. Canahuate, "Hybrid query optimization for hard-to-compress bit-vectors", *The VLDB Journal*, vol. 25, no. 3, pp. 339–354, Jun. 2016, ISSN: 1066-8888. DOI: 10.1007/s00778-015-0419-9.

[66] R. W. Hamming, "Error detecting and error correcting codes", *Bell System Technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[67] J. Henkel, "Emerging memory technologies", *IEEE Design & Test*, vol. 34, no. 3, pp. 4–5, 2017.

[68] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. R. Nassif, M. Shafique, M. B. Tahoori, and N. Wehn, "Reliable on-chip systems in the nano-era: Lessons learnt and future trends", in *DAC*, 2013, 99:1–99:10.

[69] M. Hoffmann, P. Ulbrich, C. Dietrich, H. Schirmeier, D. Lohmann, and W. Schröder-Preikschat, "A Practitioner's Guide to Software-Based Soft-Error Mitigation Using AN-Codes", in *Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE '14)*, I. C. Society, Ed., Miami, FL, USA, 2014, pp. 33–40, ISBN: 978-1-4799-3465-2. DOI: 10.1109/HASE.2014.14.

[70] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the nonstop cyclone processor", *ACM SIGARCH Computer Architecture News*, vol. 18, no. 2SI, pp. 216–226, 1990.

[71] M.-Y. Hsiao, "A class of optimal minimum odd-weight-column sec-ded codes", *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[72] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design", *SIGARCH Comput. Archit. News*, vol. 40, no. 1, 2012, ISSN: 0163-5964.

[73] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten, "Monetdb: Two decades of research in column-oriented database architectures", *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 40–45, 2012.

[74] Intel Corporation. (Nov. 2016). Intel intrinsics guide, [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[75] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results", in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '91, Denver, Colorado, USA: ACM, 1991, pp. 268–277, ISBN: 0-89791-425-2. DOI: 10.1145/115790.115835.

[76] B. R. Iyer and D. Wilhite, "Data compression support in databases", in *VLDB*, vol. 94, 1994, pp. 695–704.

[77] L. Jiang, Y. Zhang, and J. Yang, "Mitigating write disturbance in super-dense phase change memories", in *DSN*, 2014, pp. 216–227.

[78] T. Karnagel, D. Habich, and W. Lehner, "Adaptive work placement for query processing on heterogeneous computing resources", *PVLDB*, vol. 10, no. 7, pp. 733–744, 2017.

[79] S. M. Khan, D. Lee, and O. Mutlu, “PARBOR: an efficient system-level technique to detect data-dependent failures in DRAM”, in *DSN*, 2016, pp. 239–250.

[80] S. Khan, D. Lee, Y. Kim, A. R. Alameldeen, C. Wilkerson, and O. Mutlu, “The efficacy of error mitigation techniques for dram retention failures: A comparative experimental study”, *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 1, pp. 519–532, Jun. 2014.

[81] J. Kim, N. Hardavellas, K. Mai, B. Falsafi, and J. Hoe, “Multi-bit error tolerant caches using two-dimensional error coding”, in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, Washington, DC, USA: IEEE Computer Society, 2007, pp. 197–209, ISBN: 0-7695-3047-8. DOI: 10.1109/MICRO.2007.28.

[82] Y. Kim, R. Daly, J. Kim, C. Fallin, J. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, “Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors”, in *ISCA*, 2014, pp. 361–372.

[83] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, “Kiss-tree: Smart latch-free in-memory indexing on modern architectures”, in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, ser. DaMoN ’12, New York, NY, USA: ACM, 2012, pp. 16–23, ISBN: 978-1-4503-1445-9. DOI: 10.1145/2236584.2236587.

[84] D. E. Knuth, *The Art of Computer Programming, Sorting and Searching*. Addison-Wesley, Reading, MA, 1973, vol. 3.

[85] T. Kolditz, D. Habich, P. Damme, W. Lehner, D. Kuvaiskii, O. Oleksenko, and C. Fetzer, “Resiliency-aware data compression for in-memory database systems”, in *DATA 2015 - Proceedings of 4th International Conference on Data Management Technologies and Applications, Colmar, Alsace, France, 20-22 July, 2015.*, 2015, pp. 326–331. DOI: 10.5220/0005557303260331.

[86] T. Kolditz, D. Habich, D. Kuvaiskii, W. Lehner, and C. Fetzer, “Needles in the haystack — tackling bit flips in lightweight compressed data”, in *Data Management Technologies and Applications: 4th International Conference, DATA 2015, Colmar, France, July 20-22, 2015, Revised Selected Papers*, M. Helfert, A. Holzinger, O. Belo, and C. Francalanci, Eds. Springer International Publishing, 2016, pp. 135–153, ISBN: 978-3-319-30162-4. DOI: 10.1007/978-3-319-30162-4_9.

[87] T. Kolditz, D. Habich, W. Lehner, M. Werner, and S. T. de Bruijn, “AHEAD: Adaptable data hardening for on-the-fly hardware error detection during database query processing”, in *SIGMOD/PODS ’18: 2018 International Conference on Management of Data*, (Houston, TX, USA), New York, NY, USA: ACM, Jun. 10–15, 2018. DOI: 10.1145/3183713.3183740.

[88] T. Kolditz, T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, “Online bit flip detection for in-memory b-trees on unreliable hardware”, in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, ser. DaMoN ’14, Snowbird, Utah, 2014, 5:1–5:9, ISBN: 978-1-4503-2971-2. DOI: 10.1145/2619228.2619233.

[89] T. Kolditz, B. Schlegel, D. Habich, and W. Lehner, “Online bit flip detection for in-memory b-trees live!”, in *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, 2015, pp. 675–678, ISBN: 978-3-88579-635-0. [Online]. Available: http://subs.emis.de/LNI/Proceedings/Proceedings241/article3.html.

[90] P. Koopman, K. Driscoll, and B. Hall, “Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity”, 2015.

[91] E. Kultursay, M. T. Kandemir, A. Sivasubramaniam, and O. Mutlu, “Evaluating STT-RAM as an energy-efficient main memory alternative”, in *ISPASS*, 2013, pp. 256–267.

[92] K. Küspert, “Efficient Online Error Detection Techniques for Trees in Database Systems”, English, in *Fehlertolerierende Rechensysteme*, ser. Informatik-Fachberichte, vol. 84, 1984, ISBN: 978-3-540-13348-3. DOI: 10.1007/978-3-642-69698-5_8.

[93] D. Kuvaiskii and C. Fetzer, “Δ-encoding: Practical encoded processing”, in *Dependable Systems and Networks (DSN), 2015 45th Annual IEEE/IFIP International Conference on*, IEEE, 2015, pp. 13–24.

[94] D. Kuvaiskii, O. Oleksenko, P. Bhatotia, P. Felber, and C. Fetzer, “Elzar: Triple modular redundancy using intel avx (practical experience report)”, in *DSN*, 2016, pp. 646–653.

[95] T. Lahiri, M.-A. Neimat, and S. Folkman, “Oracle timesten: An in-memory database for enterprise applications”, *IEEE Data Engineering Bulletin*, vol. 36, no. 2, p. 6, 2013.

[96] T. Lahiri, M. Neimat, and S. Folkman, “Oracle timesten: An in-memory database for enterprise applications”, *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 6–13, 2013.

[97] C. Lange and A. Ahrens, “On the undetected error probability for shortened hamming codes on channels with memory”, *Cryptography and Coding*, pp. 9–19, 2001.

[98] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, “Architecting phase change memory as a scalable dram alternative”, in *ISCA*, 2009, pp. 2–13.

[99] J. Lee, G. K. Attaluri, R. Barber, N. Chainani, O. Draese, F. Ho, S. Idreos, M. Kim, S. Lightstone, G. M. Lohman, K. Morfonios, K. Murthy, I. Pandis, L. Qiao, V. Raman, V. K. Samy, R. Sidle, K. Stolze, and L. Zhang, “Joins on encoded and partitioned data”, *PVLDB*, vol. 7, no. 13, pp. 1355–1366, 2014. DOI: 10.14778/2733004.2733008.

[100] T. J. Lehman and M. J. Carey, “A study of index structures for main memory database management systems”, in *Proceedings of the 12th International Conference on Very Large Data Bases*, ser. VLDB ’86, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 294–303, ISBN: 0-934613-18-4. [Online]. Available: http://dl.acm.org/citation.cfm?id=645913.671312.

[101] ——, “Query processing in main memory database management systems”, in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’86, Washington, D.C., USA: ACM, 1986, pp. 239–250, ISBN: 0-89791-191-1. DOI: 10.1145/16894.16878.

[102] V. Leis, A. Kemper, and T. Neumann, “The adaptive radix tree: Artful indexing for main-memory databases”, in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, Apr. 2013, pp. 38–49. DOI: 10.1109/ICDE.2013.6544812.

[103] C. Lemieux, “Monte Carlo and Quasi-Monte Carlo Sampling”, in, Springer, Ed. 2009, ISBN: 978-1441926760.

[104] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization”, *Software: Practice and Experience*, vol. 45, no. 1, pp. 1–29, 2015, ISSN: 1097-024X. DOI: 10.1002/spe.2203.

[105] D. Lemire and L. Boytsov, “Decoding billions of integers per second through vectorization”, *Softw., Pract. Exper.*, vol. 45, no. 1, pp. 1–29, 2015.

[106] J. J. Levandoski, D. B. Lomet, and S. Sengupta, “The BW-tree: A b-tree for new hardware platforms”, in *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, Apr. 2013, pp. 302–313. DOI: 10.1109/ICDE.2013.6544834.

[107] F. Li, S. Das, M. Syamala, and V. R. Narasayya, “Accelerating relational databases by leveraging remote memory and RDMA”, in *SIGMOD*, 2016, pp. 355–370.

[108] Y. Li and J. M. Patel, "Bitweaving: Fast scans for main memory data processing", in *SIGMOD*, 2013, pp. 289–300.

[109] J. H. v. Lint, *Introduction to coding theory*, 3., revised and expanded ed. Berlin; Heidelberg [u.a.]: Springer, 1999, ISBN: 9783540641339. [Online]. Available: `http://slubdd.de/katalog?TN_libero_mab21331769`.

[110] J. Liu, B. Jaiyen, Y. Kim, C. Wilkerson, and O. Mutlu, "An experimental study of data retention behavior in modern dram devices: Implications for retention time profiling mechanisms", *SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 60–71, Jun. 2013.

[111] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability", *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[112] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey", *IEEE Transactions on Computers*, vol. 37, no. 2, pp. 160–174, Feb. 1988, ISSN: 0018-9340. DOI: `10.1109/12.2145`.

[113] J. L. Massey, "Survey of residue coding for arithmetic errors", *International Computation Center Bulletin*, vol. 3, no. 4, pp. 3–17, 1964.

[114] ——, "Deep-space communications and coding: A marriage made in heaven", in *Advanced Methods for Satellite and Deep Space Communications*, Springer, 1992, pp. 1–17.

[115] T. C. Maxino and P. J. Koopman, "The effectiveness of checksums for embedded control networks", *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, pp. 59–72, Jan. 2009, ISSN: 1545-5971. DOI: `10.1109/TDSC.2007.70216`.

[116] T. Maxino, "The effectiveness of checksums for embedded networks", Master Thesis, Carnegie Mellon University, 2006.

[117] P. Mishra and M. H. Eich, "Join processing in relational databases", *ACM Comput. Surv.*, vol. 24, no. 1, pp. 63–113, Mar. 1992, ISSN: 0360-0300. DOI: `10.1145/128762.128764`.

[118] S. Mittal, "A survey of soft-error mitigation techniques for non-volatile memories", *Computers*, vol. 6, no. 1, p. 8, 2017.

[119] C. Mohan, "Disk read-write optimizations and data integrity in transaction systems using write-ahead logging", in *Proceedings of the Eleventh International Conference on Data Engineering*, Mar. 1995, pp. 324–331. DOI: `10.1109/ICDE.1995.380378`.

[120] T. K. Moon, "Error correction coding", *Mathematical Methods and Algorithms. Jhon Wiley and Son*, 2005.

[121] G. E. Moore, "Cramming more components onto integrated circuits", *Readings in computer architecture*, vol. 56, M. D. Hill, N. P. Jouppi, and G. S. Sohi, Eds., 2000.

[122] R. H. Morelos-Zaragoza, *The art of error correcting coding*, 2. ed. New York: Wiley, 2006, ISBN: 9780470015582. [Online]. Available: `http://slubdd.de/katalog?TN_libero_mab214051433`.

[123] W. Muła. (Sep. 2017). Ssse3: Fast popcount. first published in May 2008, revised in January 2017, [Online]. Available: `http://0x80.pl/articles/sse-popcount.html`.

[124] W. Mula, N. Kurz, and D. Lemire, "Faster population counts using AVX2 instructions", *CoRR*, vol. abs/1611.07612, 2016. [Online]. Available: `http://arxiv.org/abs/1611.07612`.

[125] O. Mutlu, "The rowhammer problem and other issues we may face as memory becomes denser", in *DATE*, 2017, pp. 1116–1121.

[126] R. Naseer and J. Draper, "Dec ecc design to improve memory reliability in sub-100nm technologies", in *Electronics, Circuits and Systems, 2008. ICECS 2008. 15th IEEE International Conference on*, IEEE, 2008, pp. 586–589.

[127] S. R. Nassif, "The light at the end of the cmos tunnel", in *ASAP 2010 - 21st IEEE International Conference on Application-specific Systems, Architectures and Processors*, Jul. 2010, pp. 4–9. DOI: `10.1109/ASAP.2010.5540756`.

[128] T. Neumann, "Efficiently compiling efficient query plans for modern hardware", vol. 4, no. 9, pp. 539–550, Jun. 2011, ISSN: 2150-8097. DOI: `10.14778/2002938.2002940`.

[129] ——, (Nov. 2016). The price of correctness, [Online]. Available: `http://databasearchitects.blogspot.de/2015/12/the-price-of-correctness.html`.

[130] E. Normand, "Single event upset at ground level", *IEEE transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, 1996.

[131] P. O'Neil, E. O'Neil, X. Chen, and S. Revilak, "The star schema benchmark and augmented fact table indexing", in *TPCTC*, R. Nambiar and M. Poess, Eds. 2009, pp. 237–252.

[132] N. Oh and E. J. McCluskey, "Low energy error detection technique using procedure call duplication", in *Proceedings of the 2001 International Symposium on Dependable Systems and Networks*, 2001.

[133] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures", *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 111–122, Mar. 2002, ISSN: 0018-9529. DOI: `10.1109/24.994926`.

[134] ——, "Error detection by duplicated instructions in super-scalar processors", *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, Mar. 2002, ISSN: 0018-9529. DOI: `10.1109/24.994913`.

[135] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid SCM-DRAM persistent and concurrent b-tree for storage class memory", in *SIGMOD*, 2016, pp. 371–386.

[136] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)", in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '88, Chicago, Illinois, USA: ACM, 1988, pp. 109–116, ISBN: 0-89791-268-3. DOI: `10.1145/50202.50214`.

[137] W. W. Peterson and D. T. Brown, "Cyclic codes for error detection", *IRE*, vol. 49, no. 1, pp. 228–235, 1961.

[138] F. M. Pittelli and H. Garcia-Molina, *Recovery in a triple modular redundant database system*. Princeton University, Department of Computer Science, 1987.

[139] J. Plaisance, N. Kurz, and D. Lemire, "Vectorized vbyte decoding", *CoRR*, vol. abs/1503.07387, 2015. eprint: `1503.07387`. [Online]. Available: `http://arxiv.org/abs/1503.07387`.

[140] H. Plattner, "A common database approach for oltp and olap using an in-memory column database", in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09, Providence, Rhode Island, USA: ACM, 2009, pp. 1–2, ISBN: 978-1-60558-551-2. DOI: `10.1145/1559845.1559846`.

[141] M. Poess and D. Potapov, "Data compression in oracle", in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB '03, Berlin, Germany: VLDB Endowment, 2003, pp. 937–947, ISBN: 0-12-722442-4. [Online]. Available: `https://dl.acm.org/citation.cfm?id=1315451.1315531`.

[142] F. J. Pollack, "New microarchitecture challenges in the coming generations of CMOS process technologies", in *Symposium on Microarchitecture*, 1999, p. 2.

[143] O. Polychroniou, A. Raghavan, and K. A. Ross, “Rethinking simd vectorization for in-memory databases”, in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’15, New York, NY, USA: ACM, 2015, pp. 1493–1508, ISBN: 978-1-4503-2758-9. DOI: `10.1145/2723372.2747645`.

[144] O. Polychroniou and K. A. Ross, “Vectorized bloom filters for advanced simd processors”, in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, ser. DaMoN ’14, New York, NY, USA: ACM, 2014, 6:1–6:6, ISBN: 978-1-4503-2971-2. DOI: `10.1145/2619228.2619234`.

[145] ——, “Efficient lightweight compression alongside fast scans”, in *Proceedings of the 11th International Workshop on Data Management on New Hardware*, ser. DaMoN’15, New York, NY, USA: ACM, 2015, 9:1–9:6, ISBN: 978-1-4503-3638-3. DOI: `10.1145/2771937.2771943`.

[146] I. Qualcomm Technologies. (Mar. 2018). Qualcomm continues gigabit LTE leadership with world’s first announced 2 gbps LTE modem, [Online]. Available: `https://www.qualcomm.com/news/releases/2018/02/14/qualcomm-continues-gigabit-lte-leadership-worlds-first-announced-2-gbps-lte`.

[147] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang, “Db2 with blu acceleration: So much more than just a column store”, *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1080–1091, Aug. 2013, ISSN: 2150-8097. DOI: `10.14778/2536222.2536233`.

[148] J. Rao and K. A. Ross, “Making b$^+$-trees cache conscious in main memory”, *SIGMOD Rec.*, vol. 29, no. 2, pp. 475–486, May 2000, ISSN: 0163-5808. DOI: `10.1145/335191.335449`.

[149] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, “A source-to-source compiler for generating dependable software”, in *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, IEEE, 2001, pp. 33–42.

[150] S. Rehman, M. Shafique, and J. Henkel, *Reliable Software for Unreliable Hardware - A Cross Layer Perspective*. Springer, 2016.

[151] S. K. Reinhardt and S. S. Mukherjee, *Transient fault detection via simultaneous multithreading*, 2. ACM, 2000, vol. 28.

[152] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, “SWIFT: software implemented fault tolerance”, in *CGO*, 2005, pp. 243–254.

[153] M. C. Ring. (Nov. 2016). MAPM, A Portable Arbitrary Precision Math Library in C, [Online]. Available: `http://www.tc.umn.edu/~ringx004/mapm-main.html`.

[154] N. A. Rink and J. Castrillon, “Trading fault tolerance for performance in an encoding”, in *Proceedings of the Computing Frontiers Conference*, ser. CF’17, New York, NY, USA: ACM, 2017, pp. 183–190, ISBN: 978-1-4503-4487-6. DOI: `10.1145/3075564.3075565`.

[155] N. A. Rink and J. Castrillon, “Improving code generation for software-based error detection”, *Proc. of REES*, vol. 15, 2015.

[156] R. L. Rivest. (Nov. 2016). The md5 message-digest algorithm, [Online]. Available: `https://tools.ietf.org/html/rfc1321`.

[157] E. Rotenberg, “Ar-smt: A microarchitectural approach to fault tolerance in microprocessors”, in *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, IEEE, 1999, pp. 84–91.

[158] M. A. Roth and S. J. Van Horn, “Database compression”, *SIGMOD Rec.*, vol. 22, no. 3, pp. 31–39, Sep. 1993, ISSN: 0163-5808. DOI: `10.1145/163090.163096`.

[159] G. Saake, K.-U. Sattler, and A. Heuer, *Datenbanken : Konzepte und Sprachen*, 3., aktualisierte u. erw. Aufl. Heidelberg: Mitp-Verl., 2008, ISBN: 3826616642. [Online]. Available: %7Bhttp://slubdd.de/katalog?TN_libero_mab21685903%7D.

[160] N. R. Saxena and E. J. McCluskey, "Dependable adaptive computing systems-the roar project", in *Systems, Man, and Cybernetics, 1998. 1998 IEEE International Conference on*, IEEE, vol. 3, 1998, pp. 2172–2177.

[161] U. Schiffel, "Hardware error detection using AN-codes", PhD thesis, Technische Universität Dresden, 2011.

[162] B. Schlegel, R. Gemulla, and W. Lehner, "Fast integer compression using simd instructions", in *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, ser. DaMoN '10, New York, NY, USA: ACM, 2010, pp. 34–40, ISBN: 978-1-4503-0189-3. DOI: 10.1145/1869389.1869394.

[163] D. K. Schroder, "Negative bias temperature instability: What do we understand?", *Microelectronics Reliability*, vol. 47, no. 6, pp. 841–852, 2007, Modelling the Negative Bias Temperature Instability, ISSN: 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2006.10.006.

[164] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems", *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 337–350, 2010.

[165] B. Schroeder, E. Pinheiro, and W.-D. Weber, "Dram errors in the wild: A large-scale field study", in *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS, New York, NY, USA: ACM, 2009, pp. 193–204, ISBN: 978-1-60558-511-6. DOI: 10.1145/1555349.1555372.

[166] M. Shafique, P. Axer, C. Borchert, J.-J. Chen, K.-H. Chen, B. Döbel, R. Ernst, H. Härtig, A. Heinig, R. Kapitza, F. Kriebel, D. Lohmann, P. Marwedel, R. Semeen, F. Schmoll, and O. Spinczyk, "Multi-layer software reliability for unreliable hardware", *it - Information Technology*, vol. 57, no. 3, pp. 170–180, 2015. DOI: 10.1515/itit-2014-1081.

[167] C. E. Shannon, "A mathematical theory of communication", *Bell Systems Technology Journal*, vol. 27, 379–423 and 623–656, 1948.

[168] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented edac protection against seus", *IEEE Transactions on reliability*, vol. 49, no. 3, pp. 273–284, 2000.

[169] E. Shmueli, R. Vaisenberg, Y. Elovici, and C. Glezer, "Database encryption: An overview of contemporary challenges and design considerations", *SIGMOD Record*, vol. 38, no. 3, pp. 29–34, 2009.

[170] G. Sivathanu, C. P. Wright, and E. Zadok, "Ensuring data integrity in storage: Techniques and applications", in *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*, ser. StorageSS '05, New York, NY, USA: ACM, 2005, pp. 26–36, ISBN: 1-59593-233-X. DOI: 10.1145/1103780.1103784.

[171] T. J. Slegel, R. M. Averill, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, *et al.*, "Ibm's s/390 g5 microprocessor design", *IEEE micro*, vol. 19, no. 2, pp. 12–23, 1999.

[172] M. Spica and T. M. Mak, "Do we need anything more than single bit error correction (ecc)?", in *MTDT*, 2004, pp. 111–116.

[173] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory errors in modern systems: The good, the bad, and the ugly", *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1, pp. 297–310, 2015.

[174] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in dram and sram faults", in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, Colorado: ACM, 2013, pp. 1–11, ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503257.

[175] A. A. Stepanov, A. R. Gangolli, D. E. Rose, R. J. Ernst, and P. S. Oberoi, "Simd-based decoding of posting lists", in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, ser. CIKM '11, New York, NY, USA: ACM, 2011, pp. 317–326, ISBN: 978-1-4503-0717-8. DOI: 10.1145/2063576.2063627.

[176] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-Store: A column-oriented DBMS", in *Proceedings of the 31st International Conference on Very Large Data Bases*, ser. VLDB '05, Trondheim, Norway: VLDB Endowment, 2005, pp. 553–564, ISBN: 1-59593-154-6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083592.1083658.

[177] M. Sullivan and M. Stonebraker, "Using write protected data structures to improve software fault tolerance in highly available database management systems", in *VLDB*, 1991, pp. 171–180.

[178] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schroder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy", in *Proceedings of 2012 Ninth European Dependable Computing Conference (EDCC)*, IEEE, 2012, pp. 49–60.

[179] T. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading", in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society, vol. 30, 2002, pp. 87–98.

[180] A. J. Viterbi, "Error bounds for convolutional codes and an asymtotically optimum decoding algorithm", vol. IT-13, pp. 260–269, 1967.

[181] J. F. Wakerly, *Error detecting codes, self-checking circuits and applications*. North Holland, 1978, vol. 3.

[182] H. S. Warren, *Hacker's delight*. Pearson Education, 2013.

[183] M. Welschenbach, *Cryptography in C and C++*, 2nd Edition. Apress, 2005, ISBN: 1590595025. [Online]. Available: http://katalogbeta.slub-dresden.de/id/0003598277/.

[184] M. Werner, T. Kolditz, T. Karnagel, D. Habich, and W. Lehner, "Multi-gpu approximation methods for silent data corruption of an codes", in *IWSBP*, 2016.

[185] M. Werner, T. Kolditz, T. Karnagel, D. Habich, and W. Lehner, "Multi-GPU Approximation for Silent Data Corruption of AN Codes", in *Further Improvements in the Boolean Domain*, B. Steinbach, Ed. Cambridge Scholars Publishing, Jan. 1, 2018, ISBN: 978-1-5275-0371-7. [Online]. Available: http://www.cambridgescholars.com/further-improvements-in-the-boolean-domain.

[186] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte, "The implementation and performance of compressed databases", *ACM SIGMOD Record*, vol. 29, no. 3, pp. 55–67, Sep. 2000, ISSN: 0163-5808. DOI: 10.1145/362084.362137.

[187] G. S. White, "Coded decimal number systems for digital computers", *Proceedings of the IRE*, vol. 41, no. 10, pp. 1450–1452, Oct. 1953, ISSN: 0096-8390. DOI: 10.1109/JRPROC.1953.274330.

[188] M. V. Wilkes, D. J. Wheeler, S. Gill, and F. Corbató, "The preparation of programs for an electronic digital computer", *Physics Today*, vol. 11, p. 28, 1958.

[189] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing database column scans with complex predicates", in *ADMS*, 2013, pp. 1–12.

[190] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "Simd-scan: Ultra fast in-memory table scan using on-chip vector processing units", *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 385–394, Aug. 2009, ISSN: 2150-8097. DOI: `10.14778/1687627.1687671`.

[191] H. P. Wong, H. Lee, S. Yu, Y. Chen, Y. Wu, P. Chen, B. Lee, F. T. Chen, and M. Tsai, "Metal-oxide RRAM", *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[192] Y. C. Yeh, "Triple-triple redundant 777 primary flight computer", in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, IEEE, vol. 1, 1996, pp. 293–307.

[193] ——, "Design considerations in boeing 777 fly-by-wire computers", in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, IEEE, 1998, pp. 64–72.

[194] W. X. Zhao, X. Zhang, D. Lemire, D. Shan, J.-Y. Nie, H. Yan, and J.-R. Wen, "A general simd-based approach to accelerating compression algorithms", *ACM Trans. Inf. Syst.*, vol. 33, no. 3, pp. 1–28, Mar. 2015, ISSN: 1046-8188. DOI: `10.1145/2735629`.

[195] J. Zhou and K. A. Ross, "Implementing database operations using simd instructions", in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '02, New York, NY, USA: ACM, 2002, pp. 145–156, ISBN: 1-58113-497-5. DOI: `10.1145/564691.564709`.

[196] D. Zhu, R. Melhem, and D. Mosse, "The effects of energy management on reliability in real-time embedded systems", in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, Nov. 2004, pp. 35–40. DOI: `10.1109/ICCAD.2004.1382539`.

[197] J. F. Ziegler and W. A. Lanford, "Effect of cosmic rays on computer memories", *Science*, vol. 206, no. 4420, pp. 776–788, 1979, ISSN: 0036-8075. DOI: `10.1126/science.206.4420.776`. eprint: `http://science.sciencemag.org/content/206/4420/776.full.pdf`.

[198] J. A. Zoutendyk, H. R. Schwartz, R. K. Watson, Z. Hasnain, and L. R. Nevill, "Single-event upset (seu) in a dram with on-chip error correction", *IEEE Transactions on Nuclear Science*, vol. 34, no. 6, pp. 1310–1315, Dec. 1987, ISSN: 0018-9499. DOI: `10.1109/TNS.1987.4337471`.

[199] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz, "Super-scalar RAM-CPU cache compression", in *ICDE*, 2006, p. 59.

[200] M. Zukowski, M. van de Wiel, and P. A. Boncz, "Vectorwise: A vectorized analytical DBMS", in *IEEE 28th International Conference on Data Engineering*, Apr. 2012, pp. 1349–1350. DOI: `10.1109/ICDE.2012.148`.

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ACRONYMS

Notation	Description	Page List
ALU	Arithmetic Logic Unit	20
AND	bit-wise AND	34
ANSI	American National Standards Institute	100, 101, 115
AST	abstract syntax tree	158
AVX	Advanced Vector Extensions	64
BAT	binary association table	136
BFW	bit flip weight b, the number of bits flipped in a data unit, e.g. a data or code word	14, 75
BSC	binary symmetric channel	4, 40
CFCSS	controlf̄low checking by software signatures	20
COTS	commercialōff̄thes̄helf	19
CPU	central processing unit	2
DDR4	double data rate-4	2
DEC	double error correction	15
DELTA	delta encoding	107
DICT	dictionary encoding	107
DMR	double modular redundancy	19
DMS	data management system	2
DRAM	dynamic random access memory	2
DSM	decomposition storage model	103
ECC	error correcting code	3
EDAC	error detection and correction	21
EDB-Tree	Error detecting B-Tree	128
EDDI	error detection by duplicated instructions	20
FOR	frame of reference	107
GPU	graphics processing unit	50
HDD	hard disk drive	3
ILP	instruction-level parallelism	20
IPC	instructions per cycle	169
IR	internal representation	135
ISA	instruction set architecture	11
LRCID	longitudinal range check	29
LSB	least significant bit	32

Notation	**Description**	**Page List**
MAL	MonetDB assembly language	136
MIPS	million integers per second	68
MSB	most significant bit	83
NS	null suppression	107
NSM	n-ary storage model	103
OID	object identifier	103
OR	bit-wise OR	34
OS	operating system	3
PAX	Partition Attribute Across	103
QEP	query execution plan	135
RAID	Redundant Array of Independent (sometimes also Inexpensive) Disks	10
RCB	Reliable Computing Base, see Definition 1	8
RCR	residue class ring	88
RID	record identifier	113
RLE	run length encoding	107
RMT	simultaneous redundant multithreading	16
SDC	silent data corruption	2
SECDED	single error correction, double error detection	3
SIMD	Single Instruction Multiple Data	26, 29
SMT	simultaneous multithreading	16
SPARC	Standards Planning and Requirements Committee, part of the American National Standards Institute (ANSI)	100, 101, 115
SQL	structured query language	101
SRAM	static random access memory	2
SSB	star schema benchmark	5
SSD	solid state disk	3
SSE	Streaming SIMD Extensions	64
SWIFT	software-implemented fault tolerance	20
TMR	triple modular redundancy	19
VOID	virtual object identifier	103
w.l.o.g	without loss of generality	103
XOR	bit-wise exclusive OR	4

LIST OF SYMBOLS

Notation	Description	Page List
$\lvert\mathbb{S}\rvert$	the largest effective bit width of all elements in set $\mathbb{S}$, see also Definition 7	30
$\llbracket x \rrbracket$	The population count of x, see Definition 8	30
A	AN code parameter	34
$\mathbb{A}$	The set of all possible values for AN coding parameter A, with $\mathbb{A} \subseteq \mathbb{N}^+$	34
$A^{\sigma}_{h,k}$	Super A for given $h = \lvert A \rvert$ and $k = \lvert\mathbb{D}\rvert$ with largest possible $d_{\mathrm{H,min}}$ for this combination	56
$\mathbb{A}^{\sigma}_{h,k}$	Set of super As for given $h = \lvert A \rvert$ and $k = \lvert\mathbb{D}\rvert$	56
$A^{\Psi}_{h,k}$	Optimal A for given $h = \lvert A \rvert$ and $k = \lvert\mathbb{D}\rvert$ and an optimality criterion Ψ	56
A_{φ}	The area under the conditional SDC probability distribution curve for an error code $\mathbb{C}$	57
b	Bit flip weight, i.e. number of bits set to one in an error pattern	14
$\beta^{\mathbb{C}}_{b}$	The upper bound of the distance distribution elements of weight b, for a given code $\mathbb{C}$	39
$\mathbb{C}$	Set of code words	34
c	Code word	34
$\mathbb{C}^{A}_{\mathbb{D}_{\Theta}}$	The set of AN code words for data type $\mathbb{D}$ and parameter A	34
c_{ε}	A corrupt code word	34
d	Data word	34
d_{H}	Hamming distance, the number of bits in which two given data or code words differ	26
$d_{\mathrm{H,min}}$	Minimal Hamming distance of a code $\mathbb{C}$	38
$\mathbb{D}$	Set of data words	34
$\mathbb{D}_{\Theta}$	The set of data words of a type Θ	34
η	The minimal bit flip weight detectable by some error code	57
$K^{\mathbb{C}}$	The distance distribution of a code $\mathbb{C}$	39
$\kappa^{\mathbb{C}}_{b}$	The elements or counts of weight b for distance distribution $K^{\mathbb{C}}$	39
$P(\mathrm{SDC}\lvert b)^{\mathbb{C}}$	The conditional SDC probability of code $\mathbb{C}$, see also $\varphi^{\mathbb{C}}_{b}$	40
$\varphi^{\mathbb{C}}_{b}$	Shortcut for the conditional SDC probability $P(\mathrm{SDC}\lvert b)^{\mathbb{C}}$	40

Notation	**Description**	**Page List**
Ψ	Optimality criterion for error codes	56
ψ	Objective function to satisfy an optimality criterion Ψ	57
Θ	Placeholder for data types	34
$\mathbb{Z}$	The set of integers	34

LIST OF DEFINITIONS

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 30th October 2018 Till Kolditz