TOWARDS RESOURCE-EFFICIENT DATA ANALYTICS

by

Aarati Kakaraparthy

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: 19 December, 2023

The dissertation is approved by the following members of the Final Oral Committee:

Jingesh M. Patel, Professor, Computer Sciences, Carnegie Mellon University

Paris Koutris, Associate Professor, Computer Sciences, University of Wisconsin-Madison

Xiangyao Yu, Assistant Professor, Computer Sciences, University of Wisconsin-Madison

Dimitris Papailiopoulos, Associate Professor, Electrical and Computer Engineering, University of Wisconsin-Madison

© Copyright by Aarati Kakaraparthy 2023

All Rights Reserved

Dedicated to my family.

ACKNOWLEDGMENTS

As I'm getting closer to the completion of my doctoral studies, I want to extend my deepest gratitude and appreciation to those who have helped me in my academic journey. It is the understanding and encouragement of multiple people that made this endeavor possible, and I would like to take this opportunity to thank them for the support they have given over the past years.

First and foremost, I would like to thank my advisor – Prof. Jignesh M. Patel. His unwavering commitment to academic excellence and passion for research have been a constant source of inspiration throughout my doctorate. He has provided me with invaluable guidance, resources, and encouragement at every step of the way, and I feel fortunate to have him as my advisor. One particular piece of his advice that continues to resonate with me is: "To be efficient, you have to focus on the most important thing first". Prof. Patel's work ethic, keen decision-making, and efficiency will always continue to inspire me.

The database group at the CS department fostered a stimulating and inclusive environment, owing much to the distinguished faculty at UW Madison. I would like to thank Prof. Paris Koutris, Prof. Xiangyao Yu, Prof. Shivaram Venkataraman, Prof. Dimitris Papailiopoulos, and Prof. Remzi Arpaci-Dusseau for their guidance and valuable time that have contributed to shaping my doctoral journey. Prof. Shivaram Venkataraman has given me advice on how to do good research on multiple occasions, and I'm thankful for the advice I have received from him. Prof. Paris Koutris and Prof. Xiangyao Yu have always provided me with helpful feedback on my research, and served as part of multiple committees during my PhD. Prof. Dimitris Papailiopoulos and Prof. Remzi Arpaci-Dusseau generously agreed to be a part of my defense and qualifying examination committees respectively, and I am thankful for the feedback they have given on my research.

A large part of my PhD was supported by a research assistantship at Microsoft. It has been a great experience collaborating with Kwanghyun Park and Brian Kroth during my time as a research assistant at Microsoft Gray Systems Lab, and I learned a lot from them. Avrilia Floratou has provided me with helpful resources and generously agreed to attend my preliminary examination talk. During my internship at the DMX group at Microsoft Research, Vivek Narasayya and Christian König were excellent mentors who guided me to do impactful industrial research.

Parts of my PhD were supported by a grant from CRISP and a David DeWitt fellowship in database systems. I am sincerely thankful to the Semiconductor Research Corporation (SRC) and UW-Madison, respectively, for this support. I would also like to thank the administration at the CS Department and International Student Services for the help they have given me on multiple difficult occasions that have allowed me to continue pursuing my PhD degree. Prof. Alan Fekete helped me during one of these difficult situations in Sydney, Australia, and for that I thank him.

My PhD experience would not be the same without my fellow students and collaborators at UW Madison. Special thanks to Shaleen Deep, Harshad Deshmukh, Yannis Chronis, and Zhiwei Fan for the help and advice they have provided me as senior students of the database group. Kevin Gaffney has been a great friend during my PhD, and I would like to thank him for the support he has provided. I enjoyed interacting and collaborating with Zubeyr Furkan Eryilmaz, Ainur Ainabekova, Yunjia Zhang, Martin Prammer, Elena Milkai, Ling Zhang, Junda Chen, and Xiating Ouyang, and I learned a lot from them.

I'm deeply grateful for the support and camaraderie I have received from my friends during my stay in Madison. The very many board game nights helped take the stress off at the end of a difficult week, and I'm thankful to Muni, Bhargav, Michael, Ravi, Vishnu, Kaushik, Neha, Ankit, and Shashank for arranging them. Friends that help you get through lowest points in life are the truest, and I'm fortunate to have had the friendship of Bhumesh and Bhavya to help me through some of the most difficult times of my PhD. Pragathi was my roommate when I first came to Madison, and I'm thankful for the help she provided me settling down in Madison.

Finally, and most importantly, I would like to thank my family. The unwavering support I have received from them is what kept me moving forward to complete my PhD program. My parents, my sisters, my brothers-in-law, my grandparents, my husband, my uncles and aunts, and my entire family are the pillars of my life, and I feel fortunate to have their support. My husband, Shantanu, has been a constant companion during my time in Madison, and I wouldn't have been able to achieve any of this without him by my side.

TABLE OF CONTENTS

Page

ABSTRACT . . . 1

1 Introduction . . . 2

1.1 The Landscape of Data Analytics . . . 2

1.2 Resource Efficiency is Key Going Forward . . . 4

1.3 Oppurtunities for Resource Efficiency . . . 4

2 Optimizing Databases by Learning Hidden Parameters of Solid State Drives . . . 6

2.1 Introduction . . . 6

2.2 Background . . . 8

2.2.1 The Hierarchical Architecture of SSDs . . . 8

2.2.2 Internal Operation of SSDs and Common Recommendations for Applications 10

2.3 Optimizing Databases for an SSD . . . 12

2.3.1 Learning SSD Parameters . . . 12

2.3.1.1 The Request Size Profile . . . 15

2.3.1.2 The Location Profile . . . 16

2.3.2 Rules to Analyze Database I/O Patterns . . . 16

2.3.2.1 Rules to Identify Sub-Optimal I/O Requests . . . 20

2.3.2.2 The Benchmark Experiment . . . 21

2.3.2.3 Analyzing the I/O behavior of B^+-Tree Indices . . . 22

2.3.2.4 Analyzing the I/O Behavior of Log Files . . . 22

Page

2.3.3 Techniques to Optimize Performance 24
2.4 Evaluation 28
2.4.1 use-hot-locations on SSD-S 28
2.4.1.1 Experimental Setup 28
2.4.1.2 Performance of SQLite3 30
2.4.1.3 Performance of MariaDB 32
2.4.1.4 Summary 32
2.4.2 write-aligned-stripes on SSD-T 33
2.4.2.1 Experimental Setup 33
2.4.2.2 Observed Performance 34
2.4.2.3 Write Amplification and Estimated Wear-out 34
2.4.3 contain-write-in-flash-page on SSD-T 36
2.5 Related Work 36
3 VIP Hashing - Adapting to Skew in Popularity of Data on the Fly 38
3.1 Introduction 38
3.2 Background 40
3.2.1 Hash Tables 40
3.2.1.1 On Properly Configuring the Hash Table 40
3.2.2 Some Probability Bounds and Theorems 41
3.3 Skewed Workload Generation with Wiscer 42
3.3.1 Overview 42
3.3.2 Experimental Configuration 44
3.4 Roofline Study 44
3.4.1 Default vs VIP Configuration 44
3.4.1.1 Motivation 44

Page

3.4.1.2 Generating the configurations using Wiscer 44
3.4.2 Impact of Increasing Skew 47
3.4.2.1 Workload 47
3.4.2.2 Results 47
3.4.3 Impact of Increasing the Load Factor 48
3.4.3.1 Workload 48
3.4.3.2 Results 48
3.5 Adapting to Popularity on-the-fly 49
3.5.1 Learning In-the-Loop is Costly 49
3.5.2 VIP Hashing 51
3.5.2.1 Overview 51
3.5.2.2 Learning & Adapting 54
3.5.2.3 Sensing & Dynamically Switch-on/off Learning 55
3.5.3 Parameters 58
3.5.3.1 Allocating the budget for learning – $\mathrm{N_L}$ *vs* $\mathrm{N_D}$ 58
3.5.3.2 Choosing N_L – how much to learn? 59
3.5.3.3 Parameters for sensing – N_s and c 59
3.6 Applications 59
3.6.1 PK-FK Hash Joins 59
3.6.1.1 Experimental Setup 60
3.6.1.2 Default vs VIP Hash Join 60
3.6.1.3 Application to Skewed TPC-H 61
3.6.2 Point Queries 61
3.6.2.1 Static Popularity 62
3.6.2.2 Popularity Churn 66
3.6.2.3 Steady State 66

Page

3.6.2.4 Read Mostly 66
3.7 Related Work 67

4 Splitting Dataframes for Memory-Efficient Data Analysis 69

4.1 Introduction 69
4.2 What is Splitting? 71
4.2.1 Definition 71
4.2.2 Generating a Split 72
4.2.3 Splitting vs Normalization 72
4.2.4 Splitting Dataframes 73
4.3 Splitting Dataframes in Ibis 73
4.3.1 The Ibis library 73
4.3.2 SplitDF 76
4.4 Generating Automatic Splits 76
4.4.1 `SplitGen`: A greedy algorithm for split schema generation 79
4.4.2 Splitting CSV files in Velox 80
4.5 Evaluation 81
4.5.1 Experimental Setup 81
4.5.2 Running notebooks on SplitDF 81
4.5.3 Splitting CSV Data with `SplitGen` 83
4.6 Related & Future Work 83

5 Conclusion 88

LIST OF REFERENCES 89

ABSTRACT

Data is exploding at an exponential rate, accompanied by an increase in consumption of resources to store and process the data. Being resource-efficient is a critical aspect of systems for data analytics. To this end, we investigate three opportunities of improving the resource efficiency arising from 1) adapting to the hardware, 2) adapting to the workload, and 3) re-evaluating the abstractions provided by the system.

For (1), we study widely used storage devices, namely solid state drives (SSDs). The internal operation and algorithms used by commercial SSDs are not readily accessible to the user, and we develop novel benchmarks to uncover an SSD's *hidden parameters*. Learning hidden parameters allows us to improve the SELECT operation performance of SQLite3 and MariaDB by 29% and 27% respectively on the Samsung 960 Evo SSD, while also increasing the lifetime of the SSD.

For (2), we study the hash table – a data structure which is widely used in systems for data analytics. Real-world workloads often have skew, i.e., some keys are accessed more frequently than others. We develop mechanisms to adapt the hash table to skew in the workload in a completely online fashion, resulting in improved cache-efficiency. The proposed technique called *VIP Hashing* reduces the end-to-end execution time of TPC-H query 9 on DuckDB by 20%.

Lastly, for (3), we focus on dataframe libraries used for data analysis. Inspired by the fundamental technique of normalization used in relational database systems, we describe a technique called splitting to improve the memory efficiency of dataframes. In our experiments, we found that notebooks running on *split dataframes* in the Ibis library observe a decrease in memory usage of 19-23% compared to running on regular dataframes.

Chapter 1

Introduction

1.1 The Landscape of Data Analytics

Data analytics is the pursuit of extracting insights from data, targeted towards answering questions like – how many? when? where? what would happen if? and so on. Organizations utilize data analytics to extract information from raw data, to make informed business decisions, propose strategies, and optimize performance.

Fig. 1.1 shows the landscape of tools used for data analytics. One large class of systems for data analytics are data warehouses. Popular data warehousing solutions such as Amazon Redshift [reda], Snowflake [DCZ+16], and Google BigQuery [big] enable extracting insights from petabytes of data, and typically have a *SQL-based* relational interface. On the other end of the spectrum, systems such as NumPy [HMvdW+20] and Matlab [The21] provide a *matrix-based* interface, and are used to perform linear algebra operations for machine learning (ML) tasks. Typically, these matrix-based systems operate on a single machine, and the amount of data processed is limited by the amount of RAM on the machine (a few 100GB). Another emerging class of solutions, constituting systems such as Pandas [pan] and Polars [pol23], provide a *dataframe* interface that facilitates both relational and linear algebraic operations on tabular data. Dataframe-based systems have been growing in popularity over the past decades [PZK+22], and different dataframe systems support different abstractions, i.e., there is no standardization of the dataframe model at present.

In order to scale to large datasets, single-node systems such as Matlab [The21] and Pandas [pan] (as Dask [das] and Modin [PML+20]) utilize frameworks for large-scale execution over clusters of machines, such as Hadoop [had], Ray [MNW+17], and Apache Spark [ZXW+16]. Alongside the core solutions for data processing, a host of associated solutions such as for data visualization, data integration, workflow management, flexible storage, data governance and model management are needed (Fig. 1.1) to realize data analytics at the enterprise-scale.

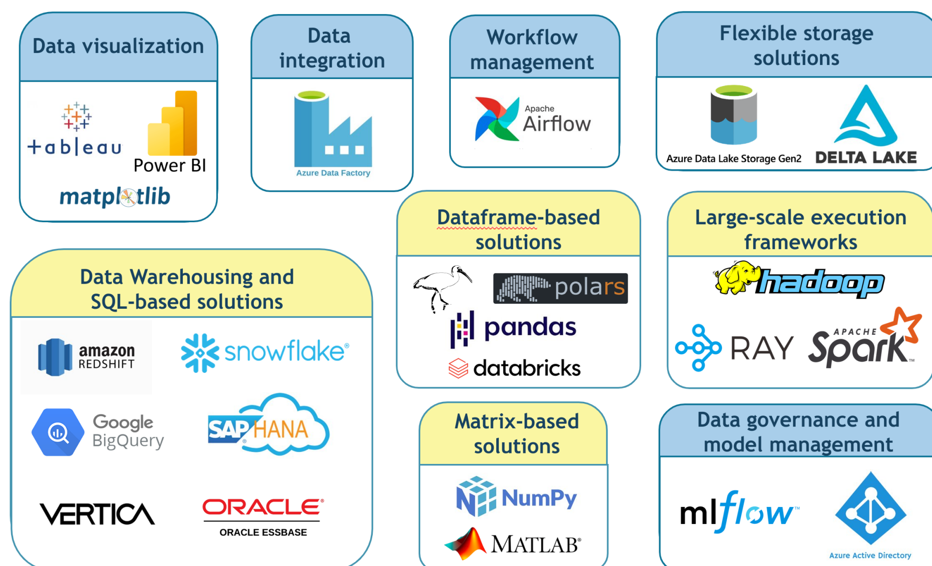


Figure 1.1: The landscape of tools for data analytics. The core solutions for data processing are highlighted in yellow.

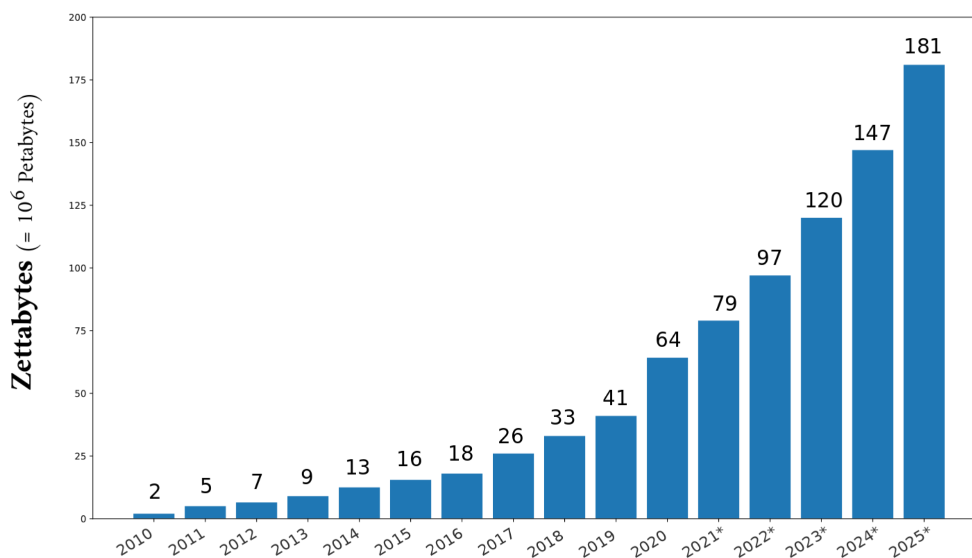


Figure 1.2: Volume of data created, copied, captured, and consumed worldwide over the years. * indicates estimated volume.

1.2 Resource Efficiency is Key Going Forward

Data is growing at an exponential scale. A recent survey [IDC20] estimated that the total volume of data in the world will cross 180 zettabytes (1 zettabyte = 1 million petabytes) in 2025 (Fig. 1.2), and this trend is expected to continue going forward.

Processing exponentially increasing data requires a corresponding increase in resources consumed and power expended. Consider the example of operating datacenters in the USA. The annual power consumption to operate these datacenters was *180 billion kWH* [Rad17] in 2022, which corresponds to about *$24 billion*. To compound the problem, power consumption is expected to increase going forward with the end of Moore's law [Sch97], and power consumption is just the tip of the iceberg in terms of resources consumed, which also include the cost of infrastructure, data movement, and human hours to name a few.

An implication of this trend is that *the impact of being resource-efficient is exponentially amplified*. Consider the previously discussed example of power consumption in data centers – a 5% improvement in performance corresponds to a significant amount of $0.05 \times \$24$ billion = $1.2 billion savings overall. Thus, resource efficiency is an important aspect of systems for data analytics, and it is going to become increasingly important going forward.

1.3 Oppurtunities for Resource Efficiency

In our work, we focus on improving the resource efficiency of systems for data analytics. As part of our work, we explore three avenues for being resource-efficient:

- **Adapting to the hardware** (Chapter 2) – We consider the case of solid state drives (SSDs) which are widely used persistent storage devices. Given that manufacturers of commercial SSDs do not reveal the internal operation and algorithms used by the device, each model of SSD has unique properties and *hidden parameters*. Learning these hidden parameters through external measurement can enable efficient utilization of the SSD to obtain better performance and/or increase the lifetime of the device. In our work [KPPK20], we developed novel benchmarks to uncover the hidden parameters of SSDs, and were able to use this information to improve the performance of database systems on commercially available SSDs.
- **Adapting to the workload** (Chapter 3) – Data structures can be made more resource-efficient by adapting to the characteristics of the workload. In our work, we consider the case of the hash table data structure, which is widely used in many systems such as relational databases, graph databases and key-value stores to name a few. Skew is a frequently encountered phenomenon in real-world workloads, where some keys are accessed more frequently

than the rest (hot vs cold keys). We show that by adapting the hash table to the skew in the workload, the data structure can be made more cache-efficient and their performance can be significantly improved. We developed *VIP Hashing* [KPKP22a], a hash table method which incorporates mechanisms to learn the skew on-the-fly and adapt the hash table configuration in a completely online manner.

- **Re-evaluating the abstractions provided by the system** (Chapter 4) – The use of dataframe-based systems has been growing over the past years owing to the popularity of performing ML tasks on tabular data. We evaluate the abstraction of dataframes to study how their structure can be made more memory-efficient. Inspired by the process of normalization utilized by relational database systems, we describe a technique called *splitting* to improve the resource efficiency of dataframes under the hood while retaining the same interface as regular dataframes. Splitting dataframes reduces redundancy in the data, and improves memory efficiency as shown in our implementation of split dataframes in Ibis.

In the subsequent chapters, we describe our work towards building resource-efficient systems by exploring each of the avenues for improvement discussed above. Through our work, we aim to highlight how improving the synergy between the system, the hardware, the workload, and user requirements can improve the resource efficiency of the system.

Chapter 2

Optimizing Databases by Learning Hidden Parameters of Solid State Drives

2.1 Introduction

Solid State Drives (SSDs) are widely used persistent storage devices, typically built with NAND Flash [SSD] and more recently with 3D XPoint memory [intb]. They are integrated circuit assemblies with no mechanical parts, and are faster than hard disks as a result [Bla, Bax, Her]. The behavior of SSDs is greatly influenced by their hierarchical architecture (see §2.2.1), and the properties of the storage medium used (see §2.2.2). For instance, a distinctive feature of SSDs is the abundance of internal parallelism, which is a consequence of their hierarchical organization.

Behind the block interface of SSDs, the internal operation and the organization of flash memory varies from one device to another. There is no silver bullet when it comes to completely determining the internal operation of an SSD. Different manufacturers use different internal policies with hidden parameters, which results in a spectrum of device characteristics. However, external measurements can be used to study these characteristics, and can sometimes reveal information regarding the internal parameters of the device.

While previous work (see §2.2.2) prescribes general guidelines to use SSDs effectively, it is possible to further optimize an application for a specific device by studying its unique characteristics. We aim to describe ways to learn hidden parameters of an SSD, and demonstrate how these parameters can be used to optimize a system for a given device. The benefits of optimizing a system for an SSD are twofold: it can improve the immediate performance of the system through better utilization of the SSD's internal parallelism, and can have long-term benefits such as increasing the lifetime of the SSD.

To motivate why optimizing for a particular SSD could be useful, consider a datacenter with a million SSDs of the same make [XSY$^+$] running a common service. Let us assume that the average lifetime of an SSD in the fleet is one year [MWKM15]. A 1% improvement in lifetime would equal

an additional 3.65 days for an individual SSD, but 3,650,000 machine days for the whole fleet. Thus, even the slightest improvement in lifetime (or performance) by optimizing the service for that particular make of SSD can cumulatively be very beneficial.

Our contribution towards optimizing a system for a given SSD consists of the following three parts (described in detail in §2.3):

1. ***Obtaining SSD characteristics and learning internal parameters*** (§2.3.1): We study two different characteristics of an SSD, namely the *request size profile* (§2.3.1.1) and the *location profile* (§2.3.1.2, [CLZ11]). From these characteristics, we learn five different internal parameters: the desirable write request sizes, the stripe size, the chunk size, the flash page size, and hot locations in the logical address space. We have performed experiments on SSDs from four different manufacturers, and the internal parameters learned have been summarized in Table 2.1.
2. ***Proposing rules to analyze the I/O patterns of a system*** (§2.3.2): Informed by the parameters learned, we propose rules (§2.3.2.1) to analyze the I/O behavior of a system when running on a particular SSD. These rules depend on the internal parameters, and thus vary between devices*. By identifying I/O calls that violate these rules, we uncover sub-optimal patterns that could potentially be corrected to improve performance.
3. ***Optimizing disk data structures to improve performance*** (§2.3.3): We examine the data structures used by the system, and describe techniques to modify them such that the sub-optimal I/O patterns found are eliminated. We present three techniques (§2.3.3), namely *use-hot-locations* (Fig. 2.10), *write-aligned-stripes* (Fig. 2.12), and *contain-write-in-flash-page* (Fig. 2.11).

In our work, we have studied two popular open-source databases, namely SQLite3 [Hip] and MariaDB (with InnoDB storage module) [Marb, inna]. While the former is a minimalistic database engine with uses spanning from mobile applications to data science platforms like pandas [pan, sqlc], the latter is widely deployed in multiple large-scale enterprises [mara]. We apply the proposed rules to study these database engines, and present techniques to optimize them on our experimental SSDs. These techniques have been evaluated in §2.4.

Applying the technique *use-hot-locations* to optimize SQLite3 and MariaDB on SSD-S increased their SELECT operation throughput by 29% and 27% respectively in the presence of memory buffering (§2.4.1). In addition to this, we benchmarked MariaDB using YCSB [CST+10], and obtained an improvement in performance ranging from 1%-22% for different workloads (Fig. 2.14b).

*We refer to a particular make of an SSD as "a device".

Table 2.1: SSDs used in experiments and their internal parameters learned. In all expressions used, i is an integer. The desirable write request sizes and the stripe size are learned from the *request size profile* (§2.3.1.1), whereas the chunk size, hot locations and the flash page size are learned from the *location profile* (§2.3.1.2).

Label	**Name**	**Interface**	**Desirable Write Request Sizes**	**Stripe Size**	**Chunk Size**	**Hot Locations**
SSD-S	Samsung 960 EVO	NVMe	32KB$\times i$	64KB	64KB	64KB$\times i$ + 32KB
SSD-I	Intel Optane 905P	NVMe	1KB$\times i$	-	4KB	4KB$\times i$
SSD-T	Toshiba XG5	NVMe	64KB$\times i$	64KB	4KB	4KB$\times i$
SSD-M	Micron M500	SATA	64KB$\times i$	64KB	4KB	4KB$\times i$

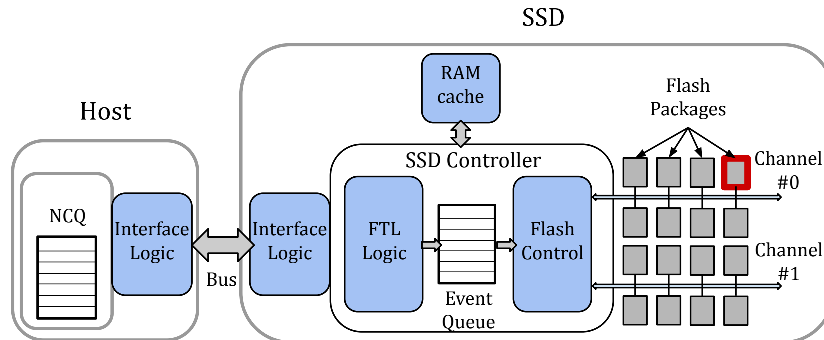
On the other hand, the techniques *write-aligned-stripes* and *contain-write-in-flash-page* reduce the device wear out caused by the log files of SQLite3 and MariaDB by 3.1% and 6.7% respectively (§2.4.2). Thus we demonstrate that the knowledge of internal parameters can be used to tune a system for a given SSD.

2.2 Background

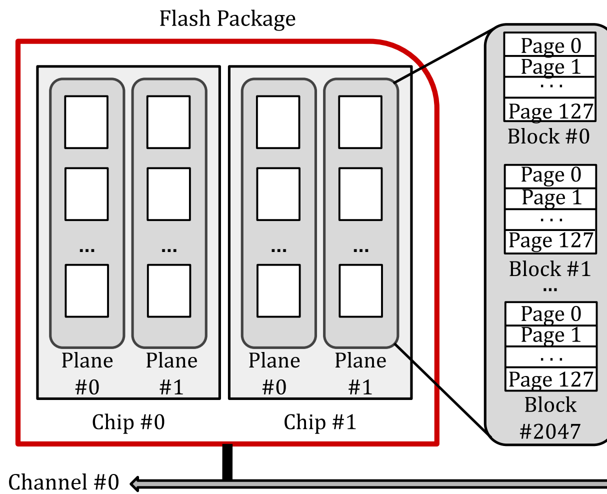
In this section, we give an overview of the hierarchical architecture of SSDs (§2.2.1), followed by an in-depth discussion on their internal operation (§2.2.2). We also discuss some common recommendations for applications informed by the general structure and behavior of SSDs.

2.2.1 The Hierarchical Architecture of SSDs

Figure 2.1 shows the architecture of an SSD. The Flash Translation Layer (FTL) is a key component responsible for managing the resources inside the SSD (Fig. 2.1a). Internally, NAND flash cells are aggregated into larger units of *flash pages* (Fig. 2.1b), which usually range from 2KB to 16KB in modern SSDs [Sam, HKADAD17, Tal]. Flash pages are further aggregated into *flash blocks*, which are typically in the order of MBs in size. Flash pages and blocks are the smallest units of read(write) and erase operations respectively inside an SSD (discussed further in §2.2.2).



(a) The architecture and internal operation of an SSD. The Host issues commands to the SSD through the interface (NVMe, SATA, etc). The FTL processes the commands and issues events to the flash control, which operates the multiple internal channels (buses). The RAM cache controlled by the FTL is used for intermediate storage during the operation of the SSD. Together, the FTL and Flash Control constitute the SSD controller.



(b) Hierarchical organization of flash memory inside a flash package. Attached to each *channel* inside an SSD are multiple *flash packages*. Each flash package has multiple *chips*, which in turn have multiple *planes*. Each plane has multiple *flash blocks*, each of which is a collection of *flash pages* (size 2KB to 16KB).

Figure 2.1: The architecture of an SSD and hierarchical organization of flash memory. The block interface is implemented by the Flash Translation Layer (FTL). Flash pages (typically 2KB to 16KB in size) and flash blocks (order of MBs) are the unit of read (write) and erase operations respectively, and are fixed for a device. Flash memory is organized hierarchically over channels, packages, chips, and planes (in that order).

This hierarchical organization of flash memory is typical to SSDs[†] (Fig. 2.1), and greatly influences their behavior. SSDs have multiple internal channels (i.e., buses) which can be operated independently, each of which is shared by multiple flash packages (also known as dies). Each flash package consists of chips, which in turn consist of multiple planes. Overall, we have four levels of internal parallelism, corresponding to each level of the storage hierarchy:

- **Channel-level**: Multiple internal channels (or buses) can be operated simultaneously and independently. Each channel is operated by at most one attached flash package at any time.
- **Package-level**: Access to all the packages on a single channel can be interleaved, and commands can be processed by packages simultaneously.
- **Chip-level & Plane-level**: Chips inside a flash package, and planes within chips can be accessed simultaneously.

The choice of physical parameters like the size of flash pages, number of channels, as well as the logic of the FTL are trade secrets of SSD manufacturers. Although the specific details may vary from one make of SSD to another, the internal operation follows similar principles informed by the properties and organization of flash memory.

2.2.2 Internal Operation of SSDs and Common Recommendations for Applications

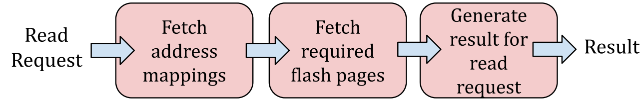
Three important low-level operations inside an SSD are *read*, *write/program*, and *erase*. Read and write operations are executed in units of *flash pages*. Unlike HDDs, flash memory is never overwritten directly; it has to be erased before writing again[‡]. The erase operation is performed in units of *flash blocks*, and data is written sequentially in an erased block in units of flash pages. The number of write-erase operations that can be performed on flash memory are limited, thus causing the SSD to *wear out* over time.

The FTL is responsible for implementing the block interface of SSDs. This requires the FTL to physically store the data onto the flash memory[§], and also maintain the the mapping of the logical block address to the physical flash address (*logical-to-physical address mappings*) where data stored.

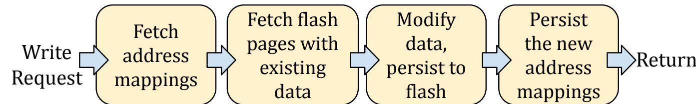
[†]Intel Optane SSD (SSD-I in Table 2.1) is not flash based. It is built with 3D XPoint Memory, but also has a hierarchical internal architecture with multiple channels [HFVW17].

[‡]However, Intel's 3D XPoint Memory need not be erased before being overwritten.

[§]Aside from managing the physical layout of data, the FTL is also responsible for various background tasks like garbage collection and wear leveling [CPP+09].



(a) Satisfying a read request. The SSD performs address translation by fetching the necessary logical-to-physical address mappings. Once the physical address(es) is(are) obtained, the flash pages containing the required data are fetched. The result for the read request is assembled and returned.



(b) Satisfying a write request. The SSD first performs address translation by fetching the necessary logical-to-physical address mappings. If the write request overwrites existing data, the flash pages containing the existing copy of the data are fetched. The updated data is assembled and written to flash memory. The address mappings corresponding to the updated data are persisted and the older mappings invalidated.

Figure 2.2: Steps performed internally by an SSD to satisfy a read and write request.

Thus, the FTL plays a critical role of deciding the physical layout of data on flash memory, which is important in determining the immediate performance of the SSD.

Figure 2.2 details the steps performed by an SSD to satisfy a read/write request. In both cases, *address translation* is performed by fetching the (logical-to-physical) address mappings corresponding to the logical address, followed by fetching the flash pages (if any) containing data. Accessing flash pages takes a significant portion of the time in satisfying a request. Hence the physical layout of data in the flash memory hierarchy, as well as the number of flash pages internally accessed is important in determining the immediate performance of an SSD (see Fig. 2.5 for an example).

SSD manufacturers have varying FTL implementations with different physical data layout policies. However, the following recommendations for applications have been made [APW+, CKZ09, CLZ11, HKADAD17, KSJ+12] based on the general operation of SSDs:

- ***Issue large or sequential write requests***. This allows the SSD to have more compact address mappings by storing information for larger logical units (*clustered pages* [CGS09, KSJ+12] instead of flash pages).
- ***Issue write requests with temporal locality***. This can result in faster address translation for subsequent read requests, as the address mappings are likely to be co-located and already present in the SSD's RAM cache.

- ***Issue large or multiple concurrent read requests***. Large read requests are likely to span over multiple sub-units inside an SSD, utilizing internal parallelism. The same holds true for multiple concurrent read requests.
- ***Issue large or multiple concurrent write requests***. This leads to better utilization of the internal parallelism of the SSD, and better performance as a result.
- ***Issue sequential read requests***. SSDs often prefetch data internally [UCH12], resulting in better performance for sequential accesses.
- ***Issue aligned write requests that are multiples of the flash page size***. For these requests, the SSD can skip fetching the existing pages being overwritten completely (step 2 in Fig. 2.2b). In contrast, requests smaller than the flash page size would need to be padded to fill the page causing *write amplification*, as more flash memory is being used than necessary. This can also cause *read amplification* for subsequent read requests.

The last recommendation requires knowledge of the flash page size, which is a *hidden* internal parameter of an SSD and is not readily shared by the manufacturers. Our aim is to go beyond generic recommendations, and make recommendations specific to a device by learning its characteristics through measurements (see §2.3.2).

2.3 Optimizing Databases for an SSD

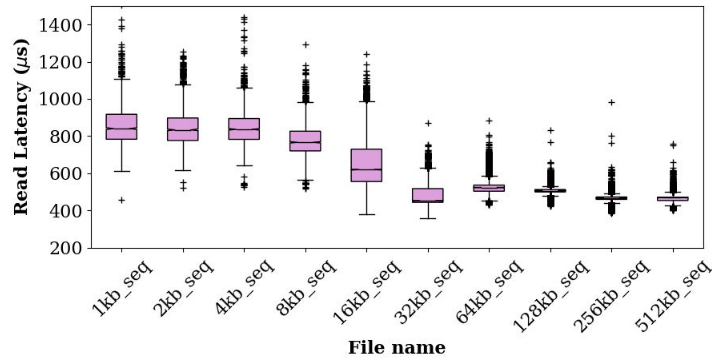
In this section, we describe our approach towards optimizing databases for an SSD. We first measure the SSD's characteristics and learn some internal parameters (§2.3.1). Informed by these parameters, we describe rules to identify any sub-optimal I/O requests issued by the database engine (§2.3.2). Finally, we propose techniques to eliminate these sub-optimal I/O requests, and improve the performance of the database engine on the SSD (§2.3.3). The SSDs used for experiments in this section, as well as their parameters learned have been summarized in Table 2.1. SSD-S, SSD-T, and SSD-M are NAND flash based, whereas SSD-I is built with 3D XPoint memory.

2.3.1 Learning SSD Parameters

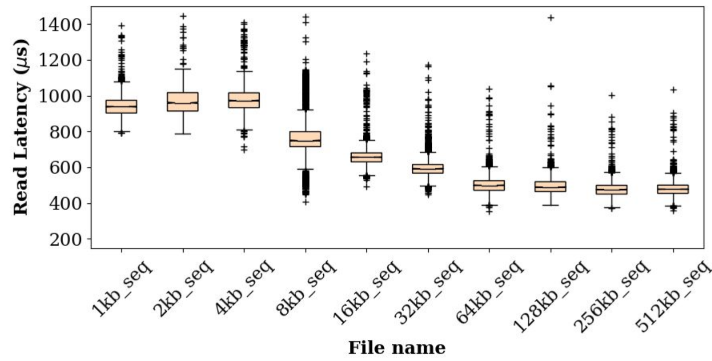
We study two types of characteristics of an SSD, which help us infer various internal parameters. First, we describe how to obtain the *request size profile* (§2.3.1.1), from which we learn the *desirable write request sizes* and the *stripe size* of an SSD. Next, we obtain the *location profile* (§2.3.1.2) of the SSDs, from which we identify their respective *chunk sizes*, *hot locations*, and *flash page sizes.*

Experiment 1 The Request Size Profile of an SSD

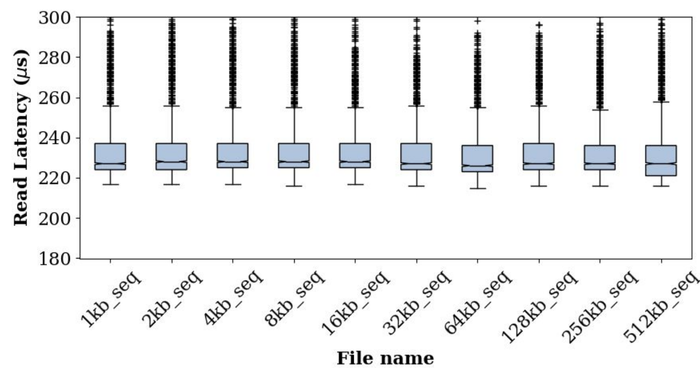
```
1: procedure GetRequestSizeProfile
2:     /* Generate files with increasing request sizes */
3:     fileSize ← 1GB
4:     filesCreated ← []
5:     numFiles ← 10
6:     for fId in 1:numFiles do
7:         requestSize ← 2^(fId−1) KB
8:         filename ← {requestSize}-seq
9:         file ← open(filename)
10:        filesCreated.append(filename)
11:        numReqs ← fileSize/requestSize
12:        for i in 1:numReqs do
13:            data ← malloc(requestSize)
14:            offset ← (i − 1) × requestSize
15:            write(file, data, offset)
16:            fsync(file)
17:
18:    // Issue read requests to the files in a random
19:    // order to avoid read-ahead inside the SSD
20:    readReqSize ← 1MB
21:    numReadReqs ← fileSize/readReqSize
22:    randOrder ← random_shuffle(1 : numReadReqs)
23:    for filename in filesCreated do
24:        file ← open(filename)
25:        latencies ← [ ]
26:        for idx in randOrder do
27:            offset ← (idx − 1) × readReqSize
28:            startTime ← time.now()
29:            data ← read(file, readReqSize, offset)
30:            endTime ← time.now()
31:            latencies.append(endTime − startTime)
32:        plot(latencies, filename)
```



(a) SSD-S. Request sizes 32KB and up are desirable. The latency becomes roughly constant after request size 64KB, indicating that it is the stripe size.

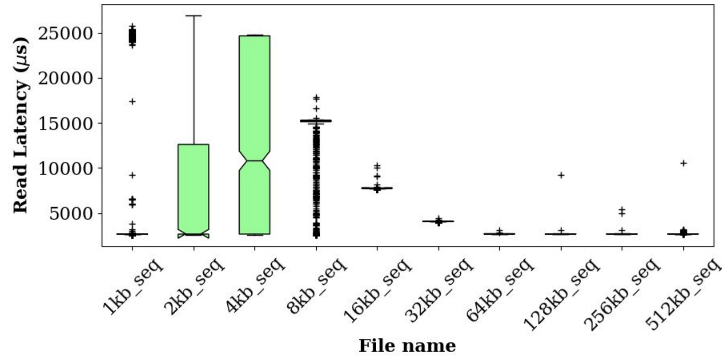


(b) SSD-T. Request sizes 64KB and up are desirable. The latency becomes roughly constant after request size 64KB, indicating that it is the stripe size.



(c) SSD-I. All request sizes are desirable. This experiment is insufficient to learn the stripe size of this SSD.

Figure 2.3: The Request Size Profile of SSD-S, SSD-T, and SSD-I. We learn the *stripe size* and *desirable request sizes* from Experiment 1.



(a) SSD-M. Request sizes 64KB and up are desirable. The latency becomes roughly constant after request size 64KB, indicating that it is the stripe size.

Figure 2.4: The Request Size Profile of SSD-M. We learn the *stripe size* and *desirable request sizes* from Experiment 1.

2.3.1.1 The Request Size Profile

Experiment 1 describes how to obtain the request size profile of an SSD. We start by creating files of size 1GB with sequential write requests of sizes ranging from 1KB to 512KB (filename *512kb-seq* refers to a 1GB file created with sequential write requests of size 512KB; likewise for other request sizes). Following this, the latency of read requests to all the files is measured. This experiment helps determine which write request sizes are desirable for a given SSD; a higher latency of read requests indicates an undesirable write request size during file creation.

Fig. 2.3 and 2.4 show the request size profile of all the SSDs, and we find that different devices have different characteristics. For a given SSD, we identify the request size at which minimum latency is attained, and all sizes greater than that, as *desirable write request sizes*. Two factors contribute to higher latency of a read request, namely read/write amplification and higher address translation time. The request size at which the lowest latency is obtained suggests absence of these factors, and is thus desirable. Thus, for SSD-S, request sizes 32KB and above are desirable, whereas for SSD-T and SSD-M, the desirable request sizes are 64KB and above.

Next, we attempt to learn the *stripe size* of the SSDs. We define the stripe size as the unit of decision of physical layout inside an SSD. Two (aligned) stripes of data have similar layout in terms of the sub-units occupied at each level of the flash hierarchy. The latency of read requests depends on the internal parallelism utilized inside the SSD, which in turn depends on the physical layout of data. Thus, similar latency of read requests indicates a similarity in physical layout of data. Therefore, for SSD-T, files *64kb-seq* to *512kb-seq* have similar physical layout, indicating that 64KB is the stripe size. By a similar argument, the stripe size of SSD-S and SSD-M is also 64KB.

Unlike the others, SSD-I has constant latency for all the files. Flash-based SSDs can incur read/write amplification, as flash pages are units of read/write operations. However, unlike flash memory, Intel's 3D XPoint memory is byte-addressable and need not be erased before overwriting, and this is perhaps the reason behind the difference in SSD-I's behavior. Although we find that none of the request sizes are particularly undesirable for SSD-I, this experiment is insufficient to learn its stripe size.

2.3.1.2 The Location Profile

We define the *chunk size* of an SSD as the amount of contiguous data stored on a single channel. Due to the hierarchical architecture of SSDs, the latency of read requests (of chunk size) can vary at different logical address locations. Fig. 2.5 shows the multiple cases that can arise. If a request internally spans over multiple channels, its latency can be relatively lower or higher depending on the flash page size and chunk size of the SSD. This is the motivation behind Experiment 2, which is inspired by Chen et al. [CLZ11].

In the experiment, we repeatedly guess a chunk size and issue read requests (of chunk size) at different logical address locations, giving rise to the location profile of an SSD (for that chunk size). We define an *offset group* as the group of logical addresses with the same relative offset into a chunk[¶]. If a significant variation in latency between different offset groups is observed, it indicates an influence of channel-level parallelism in some form.

Fig. 2.6 shows the result of this experiment. For SSD-S, we observe that for a chunk size of 64KB, offset group 32KB has 39% lower latency than offset group 0. Thus, the 32KB location group constitutes the set of *hot locations* on SSD-S. This behavior is similar to Fig. 2.5c, where channel-level parallelism helps reduce latency. On the other hand, the remaining SSDs have behavior similar to Fig. 2.5d with minimum latency at offset group 0, with a flash page size and chunk size of 4KB. We find that the reduction in latency at hot locations varies from one device to another, and this has been reported in Fig. 2.6.

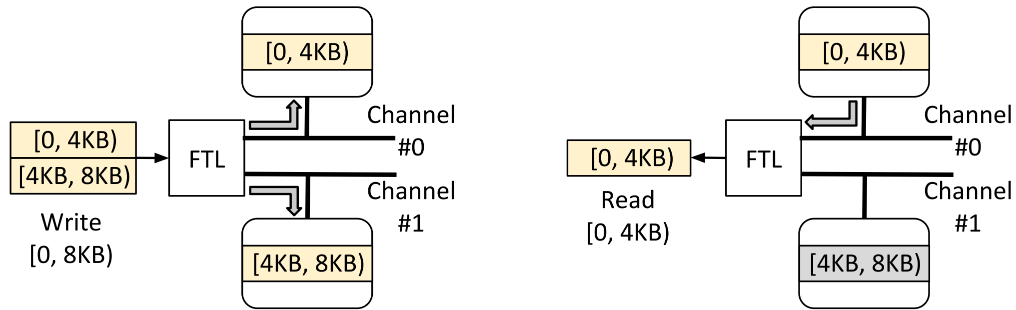
2.3.2 Rules to Analyze Database I/O Patterns

In the previous section, we described how to learn some internal parameters of an SSD. In this section, we use these parameters to propose rules for applications when running on a given SSD (§2.3.2.1).

[¶]For example, offset group 1KB for chunk size 4KB refers to all logical address locations of the form 4KB$\times i+$1KB.

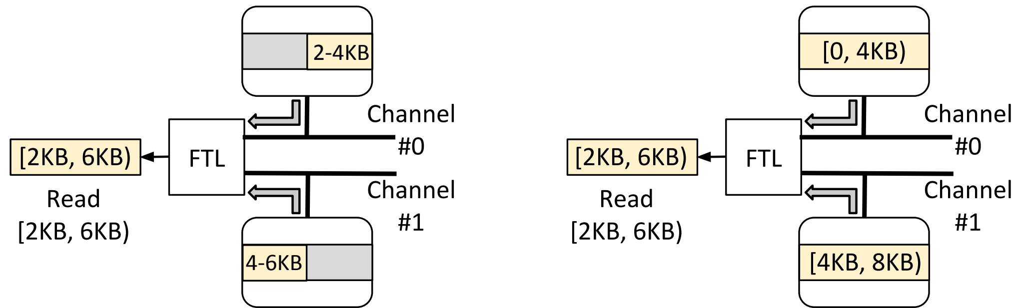
Experiment 2 The Location Profile of an SSD

```
 1: procedure GetLocationProfile
 2:     filename ← 512KB-seq
 3:     file ← open(filename)
 4:     fileSize ← 1GB
 5:     chunkSizeMin ← 4KB
 6:     chunkSizeMax ← 512KB
 7:     offsetUnit ← 1KB
 8:     expChunkSize ← chunkSizeMin
 9:     while expChunkSize <= chunkSizeMax do
10:         numChunks ← fileSize/expChunkSize
11:         randChunks ← random_shuffle(1:numChunks)
12:         numOffsetGroups ← chunkSize/offsetUnit
13:         for i in 0:(numOffsetGroups-1) do
14:             latencies ← []
15:             for j in randChunks do
16:                 chunkOffset← (j − 1) × chunkSize
17:                 offset ←chunkOffset+i× offsetUnit
18:                 startTime ← time.now()
19:                 data ← read(file, chunkSize,offset)
20:                 endTime ← time.now()
21:                 latencies.append(endTime − startTime)
22:             plot(expChunkSize, latencies, offsetGroup=i)
23:         expChunkSize ← 2 × expChunkSize
```



(a) An example of an 8KB write request to an SSD with chunk size 4KB. Two chunks of 4KB are stored on each channel.

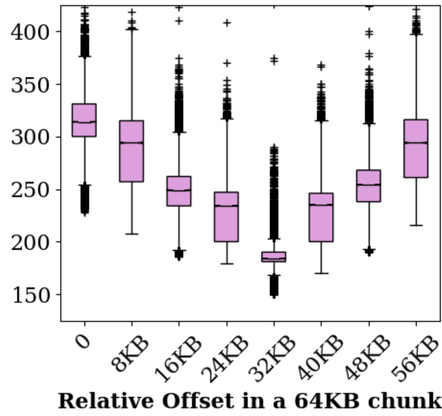
(b) A 4KB read request aligning with a chunk; data on Channel #0 alone is accessed.



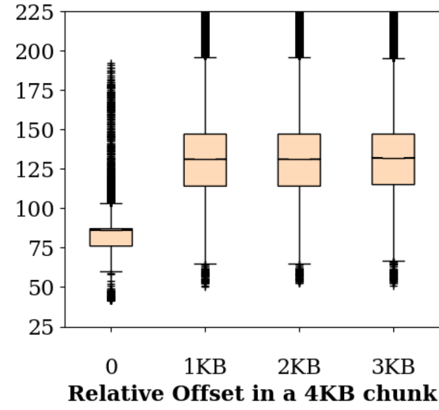
(c) A 4KB read request offset into a chunk. If the flash page size is 2KB, required pages are accessed in parallel, which will be faster than Fig. 2.5b.

(d) A 4KB read request offset into a chunk. If the flash page size is 4KB, both the chunks(pages) will be fetched. Will be slower than Fig. 2.5b.

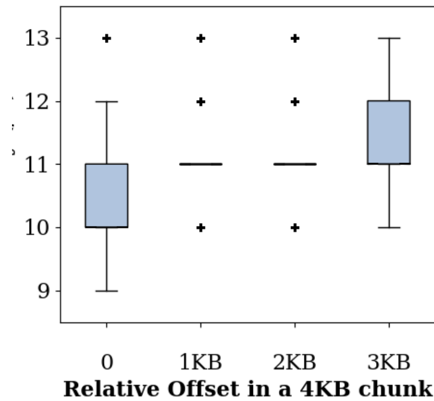
Figure 2.5: Impact of channel-level parallelism on the latency of read requests at different logical address locations. If the flash page size is smaller than the chunk size, channel-level parallelism reduces latency by doubling the bandwidth (Fig. 2.5c). Otherwise, it increases latency due to read amplification (Fig.2.5d).



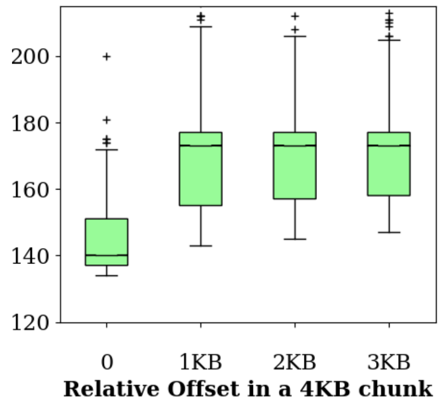
(a) **SSD-S**. Latency of reads at offset group 32KB is 39% lower than offset at group 0. The chunk size is 64KB, and the hot locations are at offset group 32KB.



(b) **SSD-T**. Latency of read requests at offset group 0 is 38% lower compared to others. The chunk size is 4KB, and the hot locations are at offset group 0.



(c) **SSD-I**. Latency of read requests at offset group 0 is 9% lower compared to others. The chunk size is 4KB, and the hot locations are at offset group 0.



(d) **SSD-M**. Latency of read requests at offset group 0 is 20% lower compared to others. The chunk size is 4KB, and the hot locations are at offset group 0.

Figure 2.6: The Location Profile. We learn the *chunk size* and *hot locations* of the SSDs from Experiment 2.

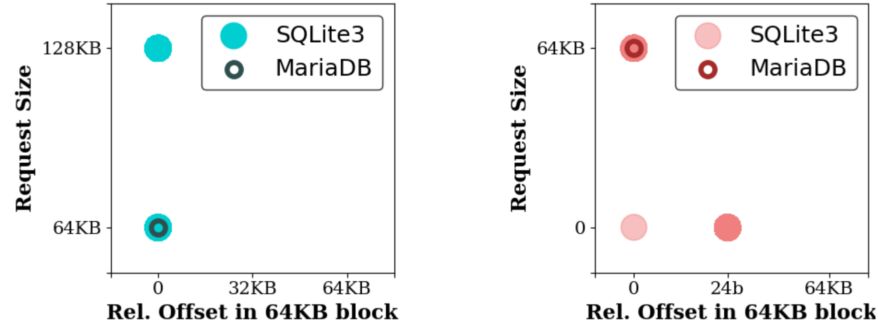
We first run a benchmark experiment (§2.3.2.2) to obtain the I/O patterns of a database engine, followed by applying the rules to analyze the I/O requests (§2.3.2.3 and §2.3.2.4). Requests that violate the rules are considered sub-optimal, and identifying such requests reveals opportunities for improving the performance of a database engine on a particular SSD. Thus, by using this approach, we study SQLite3 and MariaDB on two of our experimental SSDs, namely SSD-S and SSD-T, and we find sub-optimal I/O patterns in both of these database engines.

2.3.2.1 Rules to Identify Sub-Optimal I/O Requests

We propose the following rules for applications to follow when issuing I/O requests to a particular SSD. Requests violating these rules are considered sub-optimal.

- ***Rule 1***: *Issue write requests that are multiples of the minimum desirable write request size.* Requests not obeying this rule either cause write amplification, or require more flash memory for storing logical-to-physical address mappings.
- ***Rule 2***: *Issue read requests of chunk size at hot locations whenever possible.* Hot locations in the logical address space provide significantly lower latencies, and the application should use this advantage if it can.
- ***Rule 3***: *Issue write requests aligned with stripe boundaries, preferably of stripe size.* Overwriting stripes completely is desirable, as any existing data and address mappings can be invalidated without fetching additional flash pages. Also, unaligned requests can disturb the regularity of hot locations (discussed further in Rule 4).
- ***Rule 4***: *Do not issue unaligned write requests of chunk size.* The distribution of chunks over channels determines where hot locations occur in the logical address space. Issuing write requests not aligned with chunks can destroy the regularity of hot locations.
- ***Rule 5***: *Issue write requests such that the number of flash pages modified are minimum.* For instance, write requests smaller than the flash page size will internally be padded up to a page, and thus require at least a single flash page. However, a small write request spanning two flash pages will require modifying both the pages, causing greater write amplification than necessary, and should be avoided.

It should be noted that these rules help with maximizing the performance of the *SSD alone*. However, the requirements of the application also need to be considered for an overall improvement in performance. For instance, it would be incorrect to conclude that the database engine should always issue an I/O of 64KB (stripe size) instead of say, 16KB. Doing so might entail more I/O time than necessary and could degrade the overall performance.



(a) Write requests of sizes 64KB and 128KB are made aligned with a 64KB block (rel. offset 0). These correspond to updates to database pages.

(b) Aligned read requests of size 64KB to the database pages are made. The remaining small requests are issued by SQLite3 to access its database header fields.

Figure 2.7: I/O profile of SQLite3 and MariaDB B$^+$-Tree index files during an iteration of the benchmark experiment for a B$^+$-Tree page size of 64KB. Pages are laid out contiguously in the index file, resulting in aligned requests that are multiples of 64KB. Both the databases violate Rule 2, as read requests are not issued to hot locations on SSD-S.

2.3.2.2 The Benchmark Experiment

We run a benchmark experiment and record the I/O calls issued by the database engine for subsequent analysis using the above rules. This experiment helps isolate the insert/select operation throughput of the database engine as a measure of its performance. The experiment starts with an initial database containing 10 million key-value pairs in a single table. The key and value sizes are 32-byte and 100-byte respectively, and the initial size of the data is about 1.2GB. The experiment involves a single client running multiple iterations, with each iteration containing the following two phases:

1. Insert 50,000 new key-value pairs into the database in a random order.
2. Run 50,000 select queries generated randomly on the set of keys present in the database.

The size of the database increases by about 6MB (50,000 $\times$ 132B) during an iteration. The same workload has been run in all cases by using a fixed random seed. All the select operations are run in the same transaction, whereas the number of insert operations per transaction varies (details regarding this have been specified wherever applicable).

This experiment is similar to SQLite's benchmarking of its backend library [lsm]. However, informed by previous work [PMC17], we perform operations on a non-empty database instead. Required measurements (like measuring insert/select operation throughput, recording I/O calls using *strace* [str], etc.) are made during the experiment.

2.3.2.3 Analyzing the I/O behavior of B$^+$-Tree Indices

We study the I/O calls issued by SQLite3 and MariaDB to their primary index database files, to find any sub-optimal requests violating the rules (§2.3.2.1) on SSD-S. In both of these systems, the primary index file is structured as a B$^+$-Tree [sqlb, innb], and is divided into *database pages*. Each database page corresponds to a node in the B$^+$-tree (Fig. 2.10a), and the leaf nodes contain data rows.

We investigate the I/O behavior of both the database engines for a B$^+$-Tree (database) page size of 64KB. This matches the stripe size of most of our SSDs, which will be useful for a more comprehensive discussion involving the proposed rules. A single iteration of the benchmark experiment (§2.3.2.2) has been run with each insert operation in a separate transaction. Both database engines have been run in their default journaling modes (i.e, rollback journaling for SQLite3, and undo/redo logging in MariaDB w/InnoDB).

Fig. 2.7a shows the write requests made by both database engines to their corresponding primary index files. We see that all the write requests are issued at aligned offsets, and are a multiple of the stripe size of SSD-S (64KB). Thus, none of the rules have been violated while issuing write requests.

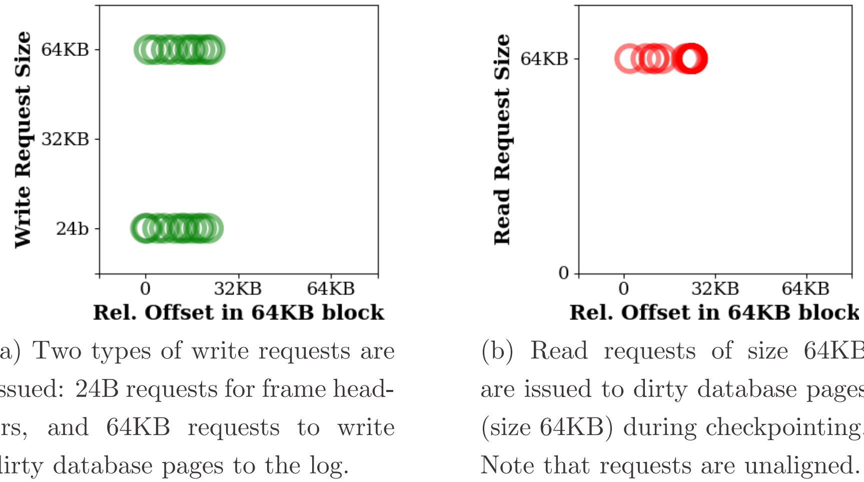
Fig. 2.7b shows the read requests issued to the B$^+$-tree index file by both the engines. The read requests of size 64KB at aligned offsets correspond to accessing pages from the index file. These requests violate Rule 2, as they are not issued to hot locations on SSD-S. The smaller read requests of size 100B and 4B are issued by SQLite3 to its database header. These requests are much smaller than a flash page and will cause read amplification. MariaDB doesn't issue any small read requests to its B$^+$-Tree database file.

2.3.2.4 Analyzing the I/O Behavior of Log Files

We consider the logging behavior of SQLite3 and MariaDB on SSD-T. We have studied SQLite3 in its write-ahead logging (WAL) mode[‖] [wal], whereas we run MariaDB in its default mode involving undo/redo logging.

Fig. 2.8a show SQLite3's write requests to the WAL file. The 64KB requests correspond to writing dirty database pages to the log, and the smaller write requests of 24B (which violate Rule 1) are for writing the log frame headers (Fig. 2.12a). Both these requests are increasingly unaligned with stripe boundaries, and violate Rule 3. All read requests (Fig. 2.8b) are of size 64KB, and are

[‖]SQLite3 in the rollback journal mode issues a mix of small and large writes to the log, similar to the WAL mode (see Fig. 2.8a). However, we only study the WAL file as a representative example.



(a) Two types of write requests are issued: 24B requests for frame headers, and 64KB requests to write dirty database pages to the log.

(b) Read requests of size 64KB are issued to dirty database pages (size 64KB) during checkpointing. Note that requests are unaligned.

Figure 2.8: I/O profile of the SQLite3 WAL log file in WAL mode during an iteration of the benchmark experiment for a B$^+$-tree page size of 64KB. The log file is organized in frames, where a frame contains a header (24B) and a dirty page (64KB). Rules 1 & 3 are violated by the write requests on SSD-T.

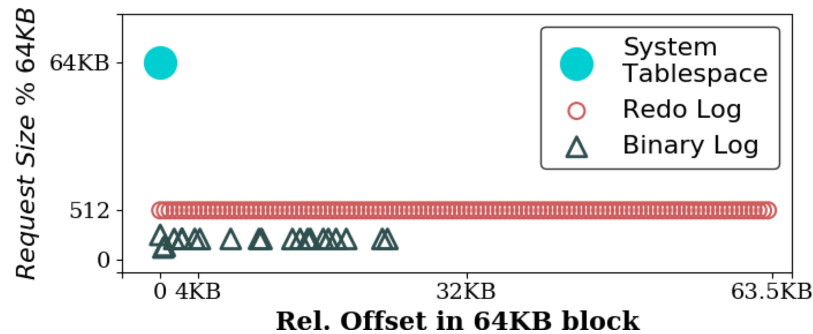


Figure 2.9: Write profile of MariaDB/InnoDB logs. The logs are contained in three files: (1) the System Tablespace file contains the undo log, doublewrite buffer, and the change buffer, (2) the Redo Log, and (3) the Binary Log for database replication. Read requests to these files follow a similar pattern. Write requests to the system tablespace file are multiples of 64KB (stripe size for SSD-S) aligning with internal stripes, and don't violate any rules. The write requests to the redo log are of size 512B (aligned with 512B boundaries), and violate Rules 1 & 3. The binary log on the other hand contains small write requests of varying sizes, which violate Rules 1, 3, and 5.

increasingly offset into the stripe boundary. These requests are not issued at a 1KB boundary, and will cause read amplification inside the SSD.

Compared to SQLite3, MariaDB has multiple log files to provide MVCC (undo logs), ensure crash recovery (redo logs) , avoid broken pages (doublewrite buffer), and provide log shipping to replicas (the binary log). These logs are contained in different tablespace files described in Fig. 2.9, and the I/O requests to these files are discussed below.

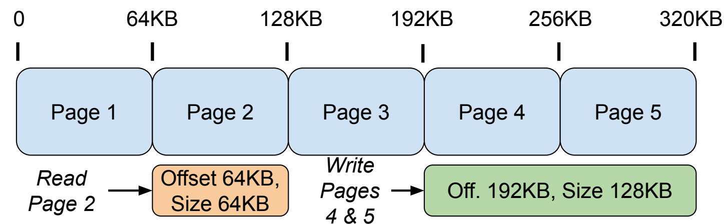
1. ***The System Tablespace (ibdata)***: Write requests of size 64KB aligning with stripe boundaries are issued, and none of the rules are violated[**].
2. ***The Redo Log (ib_logfile)***: Write requests of size 512B are issued at offsets aligning with 512B boundary (thus contained in a single flash page). Rules 1 & 3 are violated.
3. ***The Binary Log ({hostname}-bin)***: Small write requests of varying sizes at varying offsets are issued to this file. Rules 1, 3, and 5 are violated.

2.3.3 Techniques to Optimize Performance

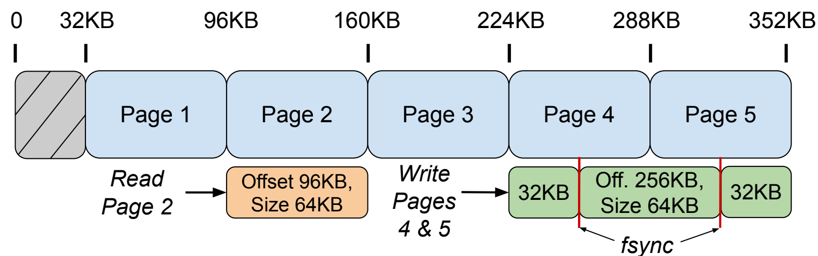
Having studied the I/O behavior of SQLite3 and MariaDB, we now propose techniques to improve their performance by eliminating sub-optimal I/O patterns. This is achieved by examining the data-structures used and making modifications to them to obtain the desired I/O behavior, while also considering the database engine's performance. We propose the following techniques:

1. ***use-hot-locations*** **on SSD-S**: Fig. 2.10a shows the original layout of the B^+-Tree database file of both SQLite3 and MariaDB, where database pages align with internal stripes of SSD-S. This violates Rule 2 (see Fig. 2.7b), as read requests are issued at stripe boundaries instead of the hot locations (32KB offset group on SSD-S). To place the pages at hot locations, we offset them by 32KB in the modified layout of the file (Fig. 2.10b). However, in order to comply with Rule 3, write requests should continue to align with stripe boundaries, as shown in Fig. 2.10b. This technique is ineffective on the remaining SSDs as their hot locations are at offset group 0 (Rule 2 isn't violated).
2. ***write-aligned-stripes*** **on SSD-T**: The SQLite3 WAL file (Fig. 2.12a) has a 32B header followed by multiple *frames*, each of which has a 24B frame header and a 64KB dirty page. These small header fields result in write amplification, as well as make the read/write requests to the dirty pages increasingly unaligned, thus violating Rules 1 & 3 (Fig. 2.8). We propose including the frame header at the end of the database page (Fig. 2.12b), and writing the WAL header (of

[**]Rule 2 would have been violated on SSD-S, but not on SSD-T as its hot locations are at offset group 0. This file's layout is similar to the B^+-Tree index file (Fig. 2.10a). However, we do not recommend applying the technique *use-hot-locations* to this file (see §2.4.1.3).

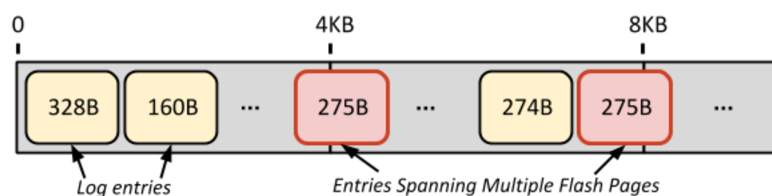


(a) The original layout of the database file. Pages are laid out contiguously, aligning with 64KB chunks. Read and write requests are issued to addresses at offset group 0.

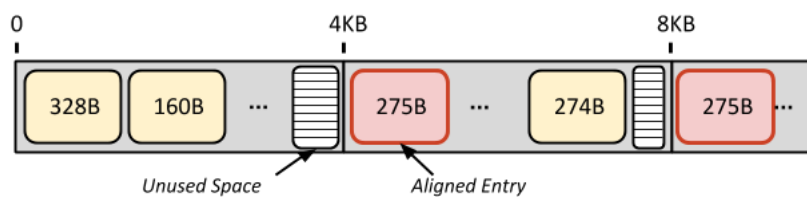


(b) The modified layout of the database file. All database pages are offset by 32KB, such that read requests to pages are issued to the 32KB offset group which are the hot locations for SSD-S. Write requests continue to align with a 64KB boundary to retain the internal chunk pattern (this is ensured by *fsync*, see §2.4.1.2).

Figure 2.10: *use-hot-locations* on SSD-S. In both SQLite3 and MariaDB, the original layout of the B$^+$-Tree database file contains contiguous aligned pages of size 64KB. Hot locations on SSD-S can be utilized if the pages are offset by 32KB. Write requests to pages need to be modified to satisfy Rule 3.

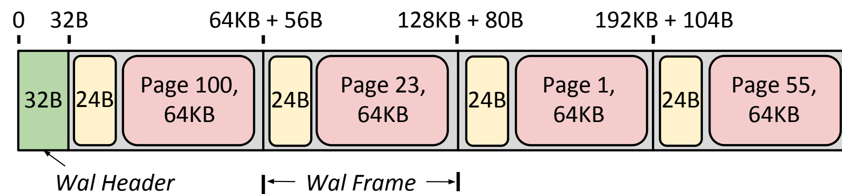


(a) The original layout of the InnoDB binary log file containing entries of varying sizes. Each entry internally requires the SSD to write a single flash page. However, entries spanning multiple flash pages (shown in red), will cause greater write amplification, as they will require modifying two flash pages instead.

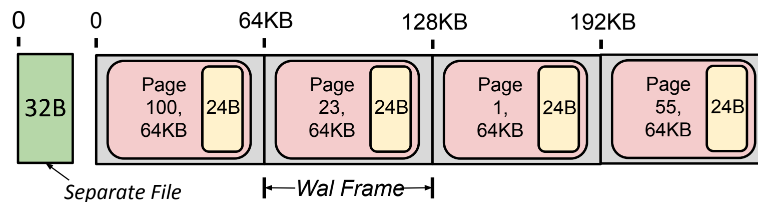


(b) The modified layout of the InnoDB binary log file. Entries spanning multiple flash pages are offset to align with the next page boundary. This results in a small varying amount of unused space at the end of each flash page, but reduces the write amplification caused by these entries to a single flash page.

Figure 2.11: *contain-write-in-flash-page* on SSD-T (flash page size 4KB). The MariaDB/InnoDB binary log file has been modified to eliminate log entries spanning over multiple flash pages. This reduces the write-amplification caused by the database engine.



(a) The original layout of the WAL log file. The file has multiple frames, each with a frame header (24B) and a dirty page (64KB). As the log frame is not a multiple of stripe size (64KB), read/write requests become increasingly unaligned and violate Rule 3 (Fig. 2.8).



(b) The modified layout of the WAL log file. The frame header is a small amount of memory, and can be included in the database page without reducing the capacity of the page by much. The new frame size is 64KB, which is equal to the stripe size. The log header of 32B is stored in a separate file to ensure that the log frames align with stripe boundaries. Writing the file header also causes write amplification, and is a one-time unavoidable cost.

Figure 2.12: *write-aligned-stripes* on SSD-T. The SQLite3 WAL log file originally contains frames of size 64KB+24B. We include the 24B frame header in the page, to issue write requests of stripe size.

32B) to a separate file. This eliminates all violations of rules on SSD-T, as shown in Fig. 2.12b. This technique will be effective on SSD-S and SSD-M as well, as they have the same stripe size as SSD-T.

3. ***contain-write-in-flash-page* on SSD-T**: The MariaDB redo log and binary log both violate Rules 1 & 3 (Fig. 2.9), as the requests are small and unaligned. However, issuing larger write requests would increase the latency of commit operation, and degrade the DB engine's performance. Additionally, the binary log also violates Rule 5, as shown in Fig. 2.11a. This violation can be eliminated by modifying the layout of the binary file as in Fig. 2.11b, where entries spanning multiple flash pages are offset to align with the next page boundary. This technique will also be effective on SSD-M, with a flash page size of 4KB.

2.4 Evaluation

We evaluate the proposed techniques, namely *use-hot-locations* (§2.4.1), *write-aligned-stripes* (§2.4.2), and *contain-write-in-flash-page* (§2.4.3). While the former technique improves the immediate performance of both the database engines on SSD-S, the latter two eliminate write amplification and increase the lifetime of SSD-T.

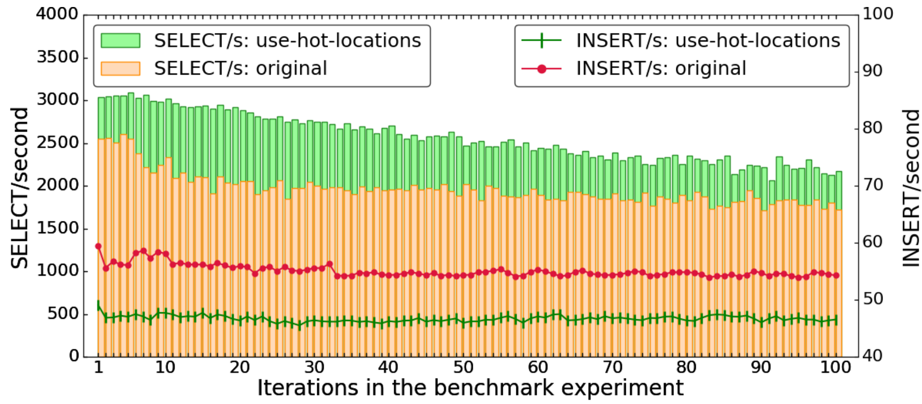
2.4.1 use-hot-locations on SSD-S

We apply the technique *use-hot-locations* (Fig. 2.10) to both SQLite3 and MariaDB, and modify the layout of the B$^+$-Tree database file as shown in Fig. 2.10b. Improvement in performance is measured as the increase in operation throughput when running the benchmark experiment (§2.3.2.2) on SSD-S. We first describe our experimental setup (§2.4.1.1), followed by discussing the performance of SQLite3 (§2.4.1.2) and MariaDB (§2.4.1.3) with the modified database file layout, and finally summarize our observations (§2.4.1.4).

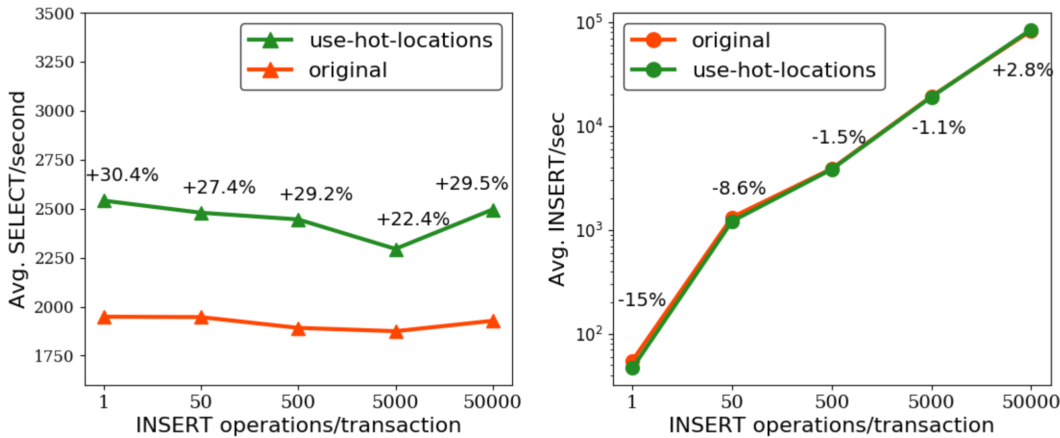
2.4.1.1 Experimental Setup

As before, SQLite3 and MariaDB have been configured with a B$^+$-Tree database page size of 64KB in their default journaling modes (rollback journaling and undo/redo logging respectively). The benchmark experiment (described in §2.3.2.2) has been run for 100 iterations, and the throughput of insert and select operations during each iteration has been measured[††]. This experiment helps us isolate and quantify the impact of *use-hot-locations*, without interference from any other factors.

[††]Throughput of insert/select operations in an iteration is measured as $\frac{\text{number of operations}}{\text{time taken in seconds}} = \frac{\text{50,000 operations}}{\text{time taken in seconds}}$.



(a) Select and Insert operation throughput over the iterations in a single run of the benchmark experiment with every insert operation in a separate transaction. Total 100 iterations are executed with 50,000 select and insert operations issued in each iteration. For select(insert) operations, we see that the performance *use-hot-locations* is consistently higher(lower) compared to the original. The average increase(decrease) in throughput is 30.4%(15.2%) over all the iterations. The loss in throughput of insert operations is caused by the additional *fsync* (Fig. 2.10b).



(b) Select and Insert operation throughput over multiple runs of the benchmark experiment with varying granularity of transactions in the insert phase. The gain/loss in throughput during each run has been indicated. For select operations, we obtain an almost constant improvement, with a median increase of 29.2%. For insert operations, as we increase the number of insert operations/transaction over the runs, the overhead of the additional *fsync* call becomes amortized and is compensated by the benefit of faster seek time. The loss in insert throughput falls under 2% from 500 insert operations/transaction onwards.

Figure 2.13: The performance of SQLite3 with *use-hot-locations* on SSD-S. The B$^+$-Tree index file is modified such that pages are stored at hot locations (Fig. 2.10b).

Theoretically, the benefit of *use-hot-locations* would be the same as the reduction in latency of read requests, i.e. 39% on SSD-S (corresponds to a 64% increase in throughput of select operations). However, applications seldom run solely on disk, and we measure the benefit of this technique in the presence of main memory. Thus, we configure the buffer pool size of SQLite3 and MariaDB to 128MB; this size is heuristically chosen to be about 10% of the initial database size before the start of the experiment (1.2GB).

Modifying the database file layout to use hot locations on SSD-S required changing the I/O modules of SQLite3 (file *os_unix.c*) and MariaDB (file *innobase/os/os0file.cc*), to offset I/O requests to the B^+-Tree index file by 32KB. Additionally, in both the database engines, the module for flushing dirty pages from the buffer pool to the database file was modified to issue aligned write requests as shown in Fig. 2.10b. Overall, we found that this technique could be incorporated with about 200 lines of code in both of these database libraries.

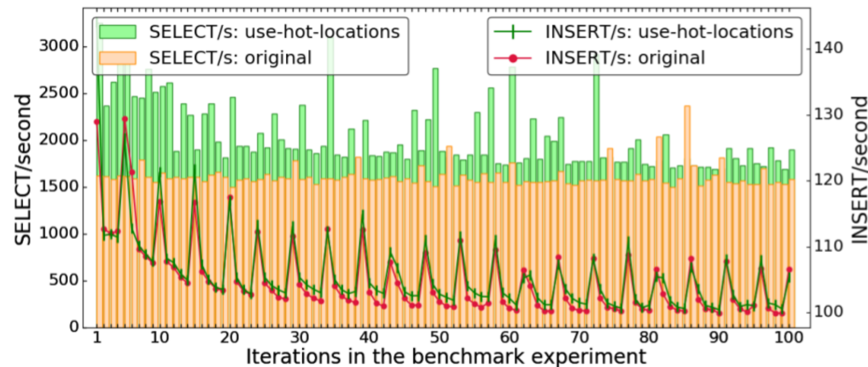
2.4.1.2 Performance of SQLite3

We run the benchmark experiment with each insert operation in a separate transaction on SQLite3 with (and without) *use-hot-locations* on SSD-S. Figure 2.13a shows the throughput of select and insert operations over multiple iterations of the benchmark experiment. On an average, we see that the throughput of select operations is 30.4% higher with *use-hot-locations* compared to the original layout. As the database pages coincide with hot locations in the modified file layout, they can be accessed faster (by 39% on SSD-S) thus reducing the latency of select operations.

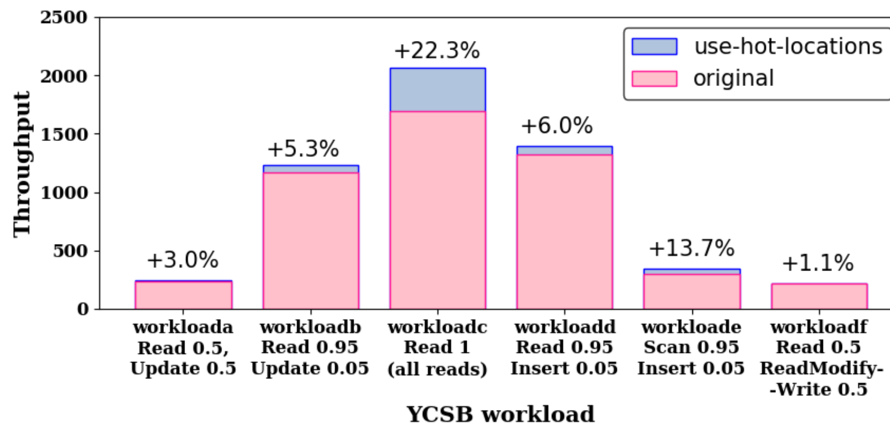
On the other hand, the throughput of insert operations is 15% lower than the original layout. The first step of an insert operation is a seek on the B^+-Tree (similar to select), which will be faster with *use-hot-locations*. However, an additional *fsync* needs to be issued at the time of commit (Fig. 2.10b), to ensure that write requests align with stripe boundaries. This *fsync*[‡‡] acts like a write barrier which guarantees that adjacent write requests are not merged by the OS; they would otherwise overwrite the stripe boundary and violate Rule 3. Thus, the overhead of the *fsync* outweighs the benefit of faster seek in this case.

To reduce the overhead of *fsync*, we measure the throughput of insert (and select) operations by running the benchmark experiment five times (Fig. 2.13b) (100 iterations are run each time) while varying the number of insert operations running in a transaction from 1 to 50,000 (i.e, varying the number transactions in the insert phase of an iteration from 50,000 to 1). Increasing the number of insert operations in a transaction (i.e., *batching* insert operations) amortizes the overhead *fsync* and

[‡‡]Only one additional fsync call is necessary. Non-adjacent write requests are not merged by the OS and can be issued together.



(a) Select and Insert operation throughput over the iterations in a single run of the benchmark experiment with each insert operation in a separate transaction. Total 100 iterations are executed with 50,000 select and insert operations issued in each iteration. For select(insert) operations, we see that the performance *use-hot-locations* is higher compared to the original. The average improvement in throughput is 26.6%(0.9%) over all the iterations. There is no loss in insert operation throughput unlike SQLite3.



(b) We run the YCSB benchmark on MariaDB with 1 million records. Each of the *workloads(a-f)* affect 100K records. We report an average performance over five runs of the benchmark, and the gain in performance for each workload has been indicated in the plot. The highest gain of 22.3% is obtained for *workloadc* (all reads), while the lowest is obtained for *workloada* and *workloadf* (3% and 1.1% respectively) as writers block readers.

Figure 2.14: The performance of MariaDB with *use-hot-locations* on SSD-S. We obtain an increase of 26.6% and 0.9% in select and insert operation throughput respectively. Unlike SQLite3 (in rollback journaling), where data is written directly to the B^+-Tree index file on commit, MariaDB (InnoDB) asynchronously copies updates to the database file using background threads, hiding the overhead of additional *fsync* (see §2.4.1.3).

improves the insert operation throughput. We find that the loss in throughput becomes less than 2% when running 500 insert operations per transaction, and ultimately, we obtain a *3% increase* in throughput when running all 50,000 insert operations in a single transaction. Also, as expected, the performance of select operations does not depend on the granularity of transactions in the insert phase, and we obtain a median increase of 29.2% in the throughput of select operations.

2.4.1.3 Performance of MariaDB

Once again, the benchmark experiment has been run with each insert operation in a separate transaction on MariaDB with (and without) *use-hot-locations* on SSD-S. Fig. 2.14a shows the throughput of select and insert operations over multiple iterations of the benchmark experiment. Similar to SQLite3, we obtain an average 26.6% increase in throughput of select operations.

However, unlike SQLite3, we do not incur any loss in insert operation throughput. This is because MariaDB (InnoDB) writes updates to the *redo log* at the time of commit, and asynchronously copies the updates to the database file using background threads. Thus, the overhead of the additional *fsync* is incurred in the background, and the observed performance of the database engine remains unchanged.

To further evaluate the benefit of *use-hot-locations* on MariaDB, we run the YCSB benchmark [CST+10] with a million database records (Fig. 2.14b). Each of the *workloads(a-f)* (refer to Fig. 2.14b for the composition of these workloads) affect 100K records of the database. The highest gain obtained is 22.3% for *workloadc* (all reads), while the lowest is obtained for *workloada* and *workloadf* (3% and 1.1% respectively) as writers seemingly block readers in both these workloads.

The System Tablespace file (Fig. 2.9) of MariaDB (InnoDB) also has a layout similar to the primary index file (Fig. 2.10a), and one might argue that we could apply *use-hot-locations* to this file as well. However, the pages from this file are seldom accessed (unless there is a transaction accessing the undo log), and using hot locations for this file would not improve the performance by much.

2.4.1.4 Summary

In conclusion, while the select operation throughput increases (by 29% and 27% for SQLite3 and MariaDB respectively) with *use-hot-locations*, the throughput of insert operations can vary depending upon how the database engine handles updates. In the case of SQLite3, the overhead of the additional *fsync* reduced the insert operation throughput by 15%, as dirty pages were written to the database file directly. However, MariaDB (InnoDB) commits any updates to the redo log instead, and the observed performance of insert operations is not affected by the *fsync*.

Table 2.2: *write-aligned-stripes* on SSD-T. While the observed performance remains the same, write amplification has been eliminated entirely by modifying the layout of the WAL log file (Fig. 2.12b).

Property	**Original**	**Optimized (write--aligned-stripes)**	**Difference (Optimized vs Original)**
Initial DB Size	19424 DB pages (of size 64KB)	19448 DB pages (of size 64KB)	+0.12%
Final DB Size (after 20 iterations)	21418 DB pages (of size 64KB)	21436 DB pages (of size 64KB)	+0.08%
Avg. Insert Operation Throughput	288 Insert/s	290 Insert/s	+0.7%
Avg. WAL checkpoint time	0.087s	0.09s	+3.4%
Avg. WAL frames written per iteration	101133	101105	+0.03%
Write amplification per iteration	50000 flash pages	0	-100%

It should be noted that the reduction in insert operation throughput of SQLite3 results from the additional *fsync* call, for lack of a better way to ensure that write requests align with stripe boundaries. This exemplifies the lack of synergy between the database system and the underlying storage, thus requiring workarounds to obtain the desired behavior. With a better contract between the system and the storage device, such an overhead could perhaps be avoided.

2.4.2 write-aligned-stripes on SSD-T

We apply the technique *write-aligned-stripes* (Fig. 2.12) to SQLite3 and modify the layout of the WAL log file as shown in Fig. 2.12b. As the log frames are modified to be of stripe size (64KB for SSD-T), we eliminate the violation of Rule 3 while writing to this file.

2.4.2.1 Experimental Setup

Again, SQLite3 has been configured with a database page size of 64KB in the write-ahead logging (WAL) mode. The benchmark experiment (§2.3.2.2) has been run for 20 iterations with a single insert operation per transaction. Auto-checkpointing in SQLite3 has been turned off (by setting *wal_autocheckpoint* to zero [pra]), and an explicit checkpoint is performed at the end of every iteration.

Various measurements are made during every iteration, and have been summarized in Table 2.2. Modifying the layout of the WAL log file required making changes to the write-ahead logging module of SQLite3 (file *wal.c*), along with reserving space at the end of every database page using `SQLITE_TESTCTRL_RESERVE` [opc] operation. Overall, this change could be incorporated with less than 100 lines of code in a highly modular manner, without affecting any other sub-modules of SQLite3.

2.4.2.2 Observed Performance

Table 2.2 shows the result of this experiment. We first discuss the directly measurable performance metrics of the database engine.

1. ***The Initial and Final DB file sizes***: As 24B of space is reserved at the end of a database page in the new layout (Fig. 2.12b), its capacity is reduced by a small amount. Thus, we see that the size of the database file is slightly larger compared to the original format.
2. ***Average throughput of Insert Operations***: We find that the throughput with the modified WAL layout is slightly higher (0.7%). This is to be expected as the size of the log frame is smaller in the modified layout, leading to faster commit operations as lesser amount of data needs to be written to the log.
3. ***Average WAL checkpoint time***: At the end of every iteration, we checkpoint the WAL log and measure the time taken to complete this operation. We find that the time taken for the checkpoint operation on an average is quite low, and is approximately the same in both cases (3.4% higher for the modified layout).
4. ***Average number of WAL frames written per iteration***: Again, we find that roughly same number of frames are written to the log in both cases. However, the objective of recording this is to estimate the amount of flash wear-out caused during the experiment.

Thus, we find that the observed performance in both cases is about the same. However, the goal of *write-aligned-stripes* is to eliminate write amplification, which we describe next.

2.4.2.3 Write Amplification and Estimated Wear-out

We measure write amplification as the number of partially written flash pages. In the original layout of the WAL log file, the 24B header causes a frame to spill over the 64KB boundary. WAL frames are written to the log file contiguously on a commit. Thus, the amount of overflow beyond the 64KB boundary on a commit in the original layout is:

$$\text{Overflow} = 24B \times \text{Number of frames written}$$

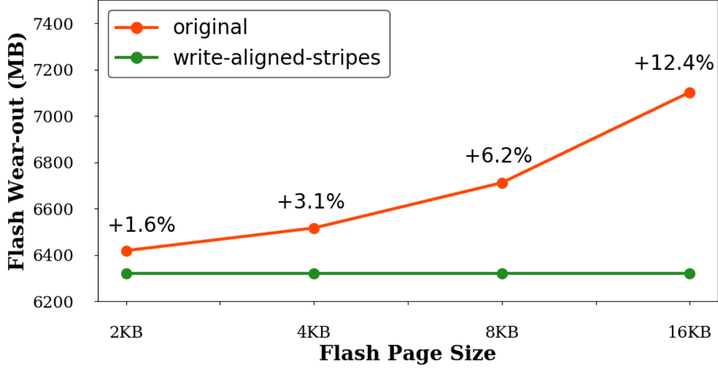


Figure 2.15: Measuring the benefit of *write-aligned-stripes* in terms of reduction in wear-out. The total flash wear-out caused during the experiment is estimated for different flash page sizes. Decreasing the wear-out corresponds to increasing the lifetime of the SSD. Thus, the increase in lifetime ranges from 1.6% to 12.4%.

In the experiment, a single insert operation has been run per transaction, and the number of frames written on a commit is low (two frames on average). Thus, the overflow is $< 100B$, which can easily be contained in a single flash page irrespective of its size (the size of a flash page ranges from 2KB to 16KB [Sam, HKADAD17, Tal]). Therefore, we report the write amplification in the original layout as 50,000 *flash pages*, as a single additional flash page is written on a commit operation. The modified layout of the WAL log file has no overflow and no write amplification as a result.

Although the write amplification can be accurately measured in units of flash pages, the total wear-out caused depends on the flash page size. We calculate the wear-out as:

$$\begin{aligned}\text{Flash wear-out} &= \text{Number of frames written} \times 64KB \\ &\quad + \text{write amplification} \times \text{flash page size}\end{aligned}$$

Fig. 2.15 shows the total flash wear-out caused during the experiment for possible flash page sizes. The difference in wear-out between the original and modified layout depends on the size of the flash page, and ranges from 1.6% to 12.4%. By reducing the wear-out, one can increase the lifetime of an SSD. Thus, the increase in the lifetime also ranges from 1.6% to 12.4%, depending on the flash page size.

This analysis holds true for any SSD, as the amount of data written to disk during the experiment remains unchanged. In particular, SSD-T has a flash page size of 4KB, and its lifetime should improve by 3.1% through this technique. Thus, transparency regarding basic parameters such as flash page size can help applications make informed decisions, while benefiting the SSD as well.

2.4.3 contain-write-in-flash-page on SSD-T

We apply the technique *contain-write-in-flash-page* (from Fig. 2.11) to the MariaDB binary log file on SSD-T. We run the benchmark experiment for 20 iterations, which results in about one million write operations to the binary log file (one write request for each insert operation's commit) of size 275B on an average. 6.7% of these write requests fall on the flash page boundary violating Rule 5. Although the actual size of the binary log is about 262MB, the total flash wear-out caused is 1.067$M \times$*flash page size(*$4KB$*)*= $4.1GB$. Thus, by offsetting these write requests to align with the next flash page boundary, we reduce the write amplification caused by the binary log by 6.7%[§§] (262MB). This technique can be applied on any SSD, by making a conservative guess (say 1KB) if the flash page size is not known.

2.5 Related Work

There is a fair amount of existing work on adapting B-Trees for flash-based SSD storage [LHY+10, AGS+09, RPK+11, JWZ+14, WKC07, NK07] based on general recommendations described in §2.2.2. A large portion of the work focuses on optimizing write operations to the SSD. Small random write operations deteriorate the performance obtained for reasons like write amplification, low bandwidth utilization, and increased address translation overhead. These can be overcome through buffering to issue large write requests [LHY+10, AGS+09, JWZ+14, WKC07], and through logging [NK07].

In [RPK+11], authors attempt to improve select performance by utilizing the internal parallelism of SSDs. This is achieved by issuing multiple read requests corresponding to multiple concurrent search operations on the B-Tree. In contrast, using hot locations reduces the latency of a single request through effective utilization of channel-level parallelism.

The internal operation of SSDs and its impact on applications has been extensively studied [PPJ+17, CLZ11, KSJ+12, HKADAD17, APW+, CKZ09]. Multiple authors have studied factors like read and write amplification [HEH+09, Wikg], the impact of address translation [HKADAD17, GKU09, APW+], performance of sequential vs. random requests [CLZ11, CKZ09, APW+, KSJ+12], and proposed rules for applications based on them. These recommendations are based on the general behavior of SSDs, and have been summarized in §2.2.2.

The hierarchical structure of SSDs [APW+, HJF+13, CLZ11, KSJ+12] and utilizing the multiple levels of parallelism [APW+, HJF+13] to improve the performance of the device have been of interest in the past. Multiple authors have taken a white box approach [APW+, HJF+13, KJKL06,

[§§] The benefit of this technique will vary, as the size of write requests to the binary log depends on the workload.

PCK[+]08, CPP[+]09] to propose how internal operations such as address translation should be handled. These studies emphasize the importance of channel-level parallelism [HJF[+]13], and have helped us build an analytical model of SSDs to reason about the performance obtained in different scenarios.

However, in the real world, commercial SSDs are a black boxes and we have attempted to learn their characteristics and parameters through measurements. Our experiment for obtaining the *request size profile* (§2.3.1.1) is informed by previous work which describes the utilization of RAID-like striping schemes [APW[+], DJ09] to distribute data at multiple levels of the storage hierarchy, whereas obtaining the *location profile* (§2.3.1.2) has been adapted from [CLZ11].

A similar approach of treating SSDs as a black box, and inferring the internal parameters through measurement has been taken in [CLZ11, KSJ[+]12]. Chen et al. [CLZ11] propose experiments to identify the chunk size and number of channels in an SSD. However, their main focus is to show the benefit of issuing concurrent requests to an SSD, and their recommendations make limited to no use of these parameters. Unlike [CLZ11], Jaehong et al. [KSJ[+]12] attempt to find the the clustered page and block size through various measurements, as well as use these parameters to tune the Linux I/O scheduler. Their objective is similar to our work, i.e., making SSD-specific optimizations informed by internal parameters. However, there are two differences between our work and [KSJ[+]12]. First, the parameters considered in our work are different from [KSJ[+]12]. Second, our recommendations for SSD-specific optimizations are at the *application-level*; for instance, techniques like *write-aligned-stripes* cannot be directly implemented in the OS, and decisions such as applying *use-hot-locations* to the primary index file alone, and not to the System Tablespace file can only be made by the application. In recent years, open-channel SSDs [BGB17] have been developed, which do not have an FTL and instead give the user full control over available resources. However, a vast majority of SSDs used are manufactured by commercial vendors, who do not reveal their internal policies. With the advent of persistent memory technologies such as 3D XPoint memory [HFVW17], one can expect hybrid devices to be developed, possibly requiring newer techniques to learn their parameters as the complexity if these devices increases.

Chapter 3

VIP Hashing - Adapting to Skew in Popularity of Data on the Fly

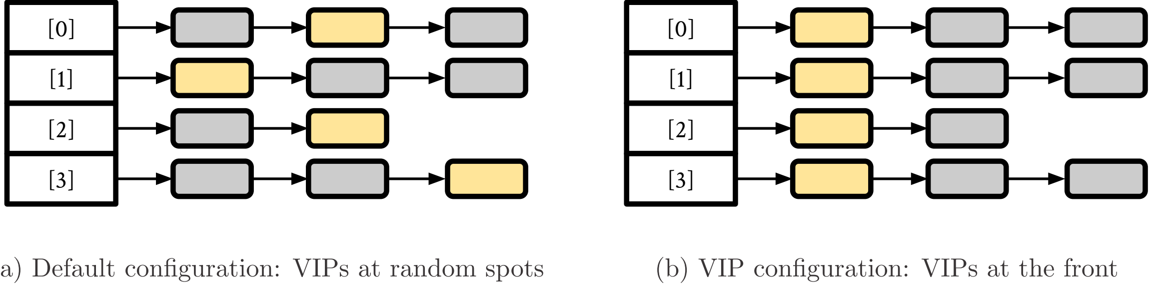
3.1 Introduction

Hash tables are widely used data structures that provide a point lookup interface – mapping a key to a value. In database systems, they are used for in-memory indexing and for query processing operations such as hash joins and aggregation. The lightweight computation involved and the constant time lookup guarantees enable hash tables to achieve high throughput when processing point queries.

However, not all keys contribute equally to the performance, and requests are often skewed towards a smaller set of "hot" keys. In multiple studies involving production workloads, fetch requests have been observed to follow the power law [Wikf, AXF+12, BCF+99] where the popularity of keys exponentially decays with the rank. The Very Important key-value Pairs (VIPs) are the keys with lower rank, as they constitute a larger portion of requests and have a greater impact on the throughput. It is possible to further improve the throughput obtained from the hash table by leveraging the skew in popularity, as we show in our work.

Fig. 3.1 shows the core motivation behind VIP Hashing – giving more favorable spots to more popular keys. In the VIP configuration (Fig. 3.1b), the keys are ordered in descending order of popularity and the VIPs are at the front, analogous to seating VIPs in the front row for an event. By placing the popular keys at the front, they can be accessed faster as they require fewer memory accesses and lesser computation (discussed in §3.4), which improves the overall throughput obtained from the hash table.

While attaining the VIP configuration is straightforward if the popularity of keys is known in advance (keys can be inserted in the right position in the chain according to their popularity), one might not have this information up front. Also, the popularity of the keys can change over time



(a) Default configuration: VIPs at random spots (b) VIP configuration: VIPs at the front

Figure 3.1: Hash Table configurations with VIP keys (in yellow) at (a) random spots, vs. (b) at the front. The throughput of the hash table can be improved by giving VIPs more favorable spots at the front of the bucket.

resulting in a different set of VIPs. Thus, more generally, one needs to learn the popularity of keys and adapt the configuration of the hash table on the fly.

It is important to note that learning requires additional computation and storage. In the case of disk-based data structures, the overhead of learning can be relatively small compared to the latency of accessing storage devices. However, hash tables are cache-sensitive data structures that perform lightweight computation, and adding overhead to hash tables can have significant impact on performance. This makes learning with hash tables notoriously challenging, as shown in §3.5.1 and prior work [SVH+21] as well. Thus, a key requirement for designing fully online learning mechanisms for hash tables is keeping the overhead in check compared to the gains.

Our contributions are as follows –

1. ***Wiscer*** (§3.3) – We developed a benchmarking tool for measuring the performance of hash tables. Wiscer can be used to generate workloads with varying levels of skew in popularity, with different ratios of fetch, insert and delete operations, and shifting hot set of keys over time. To our knowledge, no existing benchmarking tool captures all of this behavior in one place.
2. ***Roofline Analysis of the VIP configuration*** (§3.4) – We study the benefit of the VIP configuration (Fig. 3.1b) given prior knowledge of popularity. This analysis shows the maximum gain one can obtain from adapting to the skew (for a hash table with 10M keys at load factor 0.6, we observe a 57% increase in throughput in the best case), as well as the hardware trends resulting in performance gain for the VIP configuration.
3. ***Learning on a budget*** (§3.5) – We developed lightweight online mechanisms for *learning* the popularity distribution, *adapting* to the skew, *sensing* changes in the popularity distribution, and *dynamically switching on/off* the mechanisms to control the overhead. Put together, they give us the VIP Hashing method for adapting to the skew in popularity on the fly.

4. ***Application to hash joins*** (§3.6.1) – We study the application of VIP hashing to PK-FK hash joins, and we obtain a 13-23% reduction in canonical join query execution time (for a cardinality ratio of 1:16 in the relations and a hash table with load factor of 1.4). We implemented VIP hashing in DuckDB [RM19] to speed up PK-FK hash joins in single-threaded mode, and we obtain a net reduction of 20% in end-to-end execution time of TPC-H query 9 [tpc] under low skew.
5. ***Application to point queries*** (§3.6.2) – Another common use of hash tables is processing point queries. We test VIP hashing under a variety of workloads involving insert and delete operations, shifting popularity distribution of keys, different rates of shift, etc. A gain in throughput of 22% is obtained under low skew, while our choice of parameters ensures that the overhead of adapting on the fly is capped in the worst case.

Our experiments in §3.6 show that VIP hashing is a fully online non-blocking hash table method that adapts to the skew in popularity on the fly, while transparently capturing changes in the workload due to inserts, deletes, and shifting popularity distribution.

3.2 Background

3.2.1 Hash Tables

A hash table [Wika] is an associative data structure that maps keys to values. In our work, we focus on *chained hashing* (hereafter referred to as hash table). A hash table (Fig. 3.1) uses a hash function to map each key to a unique index or bucket. Since more than one key can be mapped to the same bucket, the data structure resolves these collisions by maintaining a chain (linked list) of entries belonging to the bucket. The flexibility provided by this data structure for performing insert and delete operations, along with variable length keys and values make it a popular choice in many data systems [redb, Frø, He, sqla].

3.2.1.1 On Properly Configuring the Hash Table

We focus on hashing of 8-byte integer keys and values, which is a well studied problem in past research [RAD15, BLP11]. It is important to configure the hash table correctly to draw reliable conclusions, and there are two important factors to consider. The first is the choice of the hash function. In our work, we use *MurmurHash* [App16], which is a strong hash function that provides good collision resistance in practice. The second critical aspect is the *load factor*, which is the ratio of keys to the number of buckets in the hash table. Higher load factors correspond to fewer buckets, which lead to longer chains on an average, whereas lower load factors require

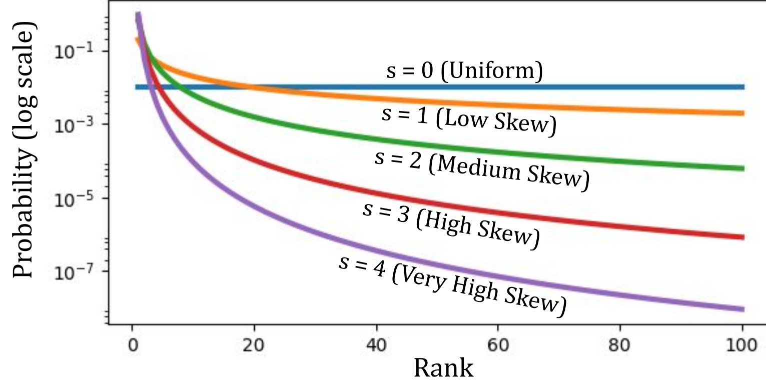


Figure 3.2: Popularity distribution of keys (number of keys $N = 100$) for different Zipfian skew factors s. Skew factor $s = 0$ corresponds to uniform popularity distribution, while $s = 1, 2, 3, 4$ simulates low, medium, high, and very high skew respectively.

more buckets and consume more memory. Informed by parameter choices in popular open-source systems [redb, He, pos], we maintain a load factor between 0.5 and 1.5 to ensure that collisions are at an acceptable level while utilizing memory efficiently. Wherever applicable, we *rehash* the hash table to maintain this range of load factor. The number of buckets in the hash table are set to be a power of two, which is a common choice [pos, He, Nat17] that speeds up the computation of the hash function. If the load factor exceeds 1.5 (falls under 0.5), we double (half) the number of buckets in the hash table.

3.2.2 Some Probability Bounds and Theorems

Below we discuss some tools related to probabilistic random variables that we use in our work.

- **Zipfian distribution**: We use Zipfian distribution [Wiki] to model varying levels of skew in fetch operations issued to keys in a hash table. Zipfian distribution has been adopted by multiple studies in the past [WSH19, BLP11, AXF+12] to statistically model skew in popularity, as it captures the power law [Wikf] characteristics of workloads that are often observed in practice [AXF+12, BCF+99].
- **Estimating mean and variance**: Let X be a random variable with mean μ and variance σ^2. Let $X_1,\ X_2, \dots X_n$ be n independent and identically distributed (i.i.d.) measurements of X. The estimated mean $\hat{\mu}$ and estimated variance $\hat{\sigma}^2$ can be evaluated as

$$\hat{\mu} = \frac{\sum_{i=1}^{n} X_i}{n}, \quad \hat{\sigma}^2 = \left(\frac{\sum_{i=1}^{n} X_i^2}{n-1} - \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n(n-1)} \right)$$

- **Gaussian tail bound confidence interval**: For a random variable X (refer above), the central limit theorem (CLT) [Wikd] states that the error in estimated mean $(\hat{\mu} - \mu)$ is approximately Gaussian distributed $\mathcal{N}(0, \frac{\sigma^2}{n})$. By applying the Gaussian pdf, a confidence interval can be obtained for the error $(\hat{\mu} - \mu)$ as follows

$$P(|\hat{\mu} - \mu| \leq t) \geq \left(1 - exp\left(\frac{-nt^2}{2\sigma^2}\right)\right) = \frac{L}{100}$$

  Thus, we can at least be $L\%$ confident that the error $|\hat{\mu} - \mu|$ is less than t. Note that the confidence increases exponentially with n (number of samples X_i drawn). It is important to note that $(\hat{\mu} - \mu)$ is only *approximately* Gaussian, so the confidence interval obtained from applying Gaussian tail bound is a heuristic.

3.3 Skewed Workload Generation with Wiscer

3.3.1 Overview

Wiscer is a benchmarking tool that we propose in our work. Wiscer has multiple configuration options (Table 3.1) that can be used to generate workloads with different levels of skew, varying proportions of fetch, insert, delete operations, different rates of popularity shift, etc. Below are some key features of Wiscer:

- **Level of skew**: Increasing levels of skew in the popularity distribution can be simulated by increasing the *zipf* factor. For instance, $zipf = 0$ and $zipf = 4$ correspond to uniform distribution and very high skew respectively (see Fig. 3.2).
- **Simulating popularity distribution shift**: The two related configuration options are *distShiftFreq* and *distShiftPrct*. After every *distShiftFreq* fetch operations, the topmost popular keys that constitute *distShiftPrct* of the requests are randomly replaced by less popular keys. This simulates a behavior where keys in the hot set become less popular after some time, which has also been observed in some real-world workloads [AXF+12].
- **Benchmarking hash table implementations**: Wiscer can optionally be used to compare different hash table implementations (option *StorageEngine*) to directly process the generated workloads without intermediate storage.
- **Fine-grained performance metrics using hardware counters**: Wiscer issues operations to the configured hash table in batches of one million requests at a time, and fine-grained metrics are collected per batch. Wiscer uses hardware counters provided by the Intel's Performance

Table 3.1: Configuration options supported by *Wiscer*

Option	Description
zipf	The zipfian factor of the popularity distribution. zipf=0 corresponds to uniform popularity.
initialSize	Initial number of keys in the hash table before running any operations.
operationCount	Total number of operations (fetch, inserts, etc.) to run on the hash table.
(fetch/insert/delete) Proportion	Proportion of operations that are fetch/insert/delete.
distShiftFreq	A shift in popularity distribution occurs after every *distShiftFreq* operations.
distShiftPrct	The popularity distribution shifts by *distShiftPrct*% every *distShiftFreq* operations.
storageEngine	Which storage engine to benchmark. Options are *ChainedHashing*, *VIPHashing*, and *none* (store workload to disk).
keyPattern	The pattern of keys to generate – *random* (default) or *sequential* (1 to n).
keyOrder	The popularity rank of keys relative to the insertion order. Options are *random* (default) and *sorted* (where keys are inserted in increasing order of popularity; a.k.a. latest).
randomSeed	The seed value (unsigned integer) to initialize the random number generator (default = 0). The random number generator is used to populate the hash table and generate the workload. Different seed values result in different instances of keys and the workload.

Monitoring Unit (PMU) [pmu] to get low-level performance metrics such as cache misses, number of cycles, retired instructions, etc.

3.3.2 Experimental Configuration

All experiments in this work are run on a Cloudlab [DRM+19] machine with two 10-core Intel Xeon Silver 4114 CPUs with a peak frequency of 3.0GHz. The server is used exclusively for running Wiscer, and the benchmarking process is pinned to a single core to avoid any overhead of context switching. The CPU scaling governor of the core has been set to *performance*, thus fixing the frequency to 3.0GHz at all times. The CPU has an L3 cache of 13.75MB, and the server machine has 192GB of RAM. This CPU belongs to the Skylake Intel architecture family [cpu17], and the PMU's hardware counters are programmed accordingly.

3.4 Roofline Study

In this section, we compare the performance of the Default and VIP configurations when the popularity of keys is static and known in advance. Since there is no overhead of learning involved in this case, this roofline study shows the maximum gain one can get from the VIP configuration for different levels of skew (§3.4.2) in popularity at different load factors (§3.4.3) of the hash table.

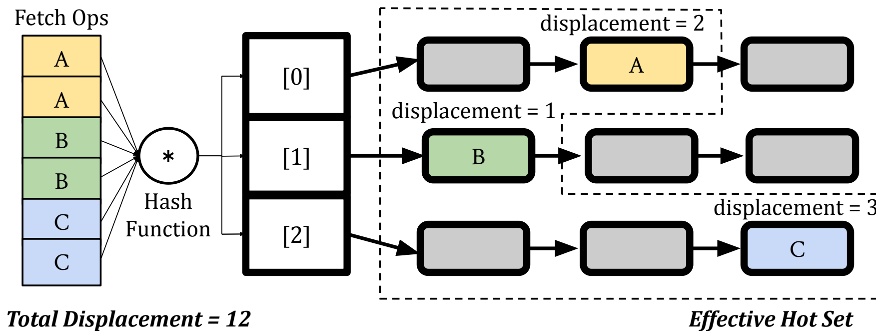
3.4.1 Default vs VIP Configuration

3.4.1.1 Motivation

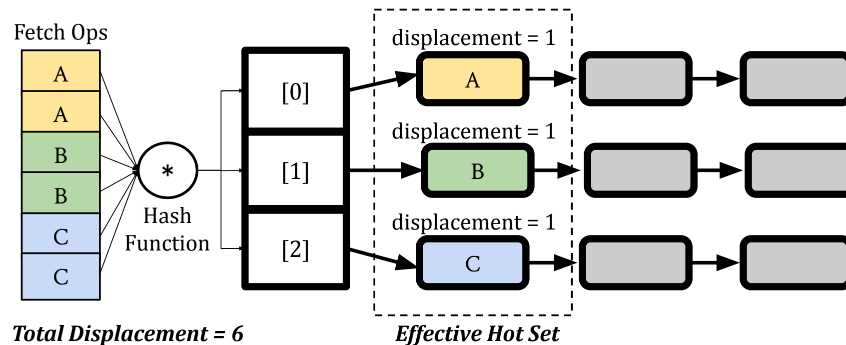
Fig. 3.3 shows an example of processing fetch requests in the Default and the VIP configurations. A key parameter to note is the *displacement* encountered, which is the total number of keys that were accessed to process the fetch requests. Accessing a key requires dereferencing a pointer and some computation. The displacement encountered in the Default configuration is higher as the less popular keys in the path to VIPs need to be accessed when processing the fetch requests and effectively become part of the hot set. A larger hot set increases the likelihood of cache misses, and we observe this trend in our experiments described next.

3.4.1.2 Generating the configurations using Wiscer

In the VIP configuration, keys in the hash table are arranged in descending order of popularity in the bucket chains (see Fig. 3.3b). We attain this configuration by running Wiscer with the default storage engine (*ChainedHashing*) and inserting keys in increasing order of popularity (*key-order=sorted*, default is *random*). Insert operations on the hash table are performed at the front of

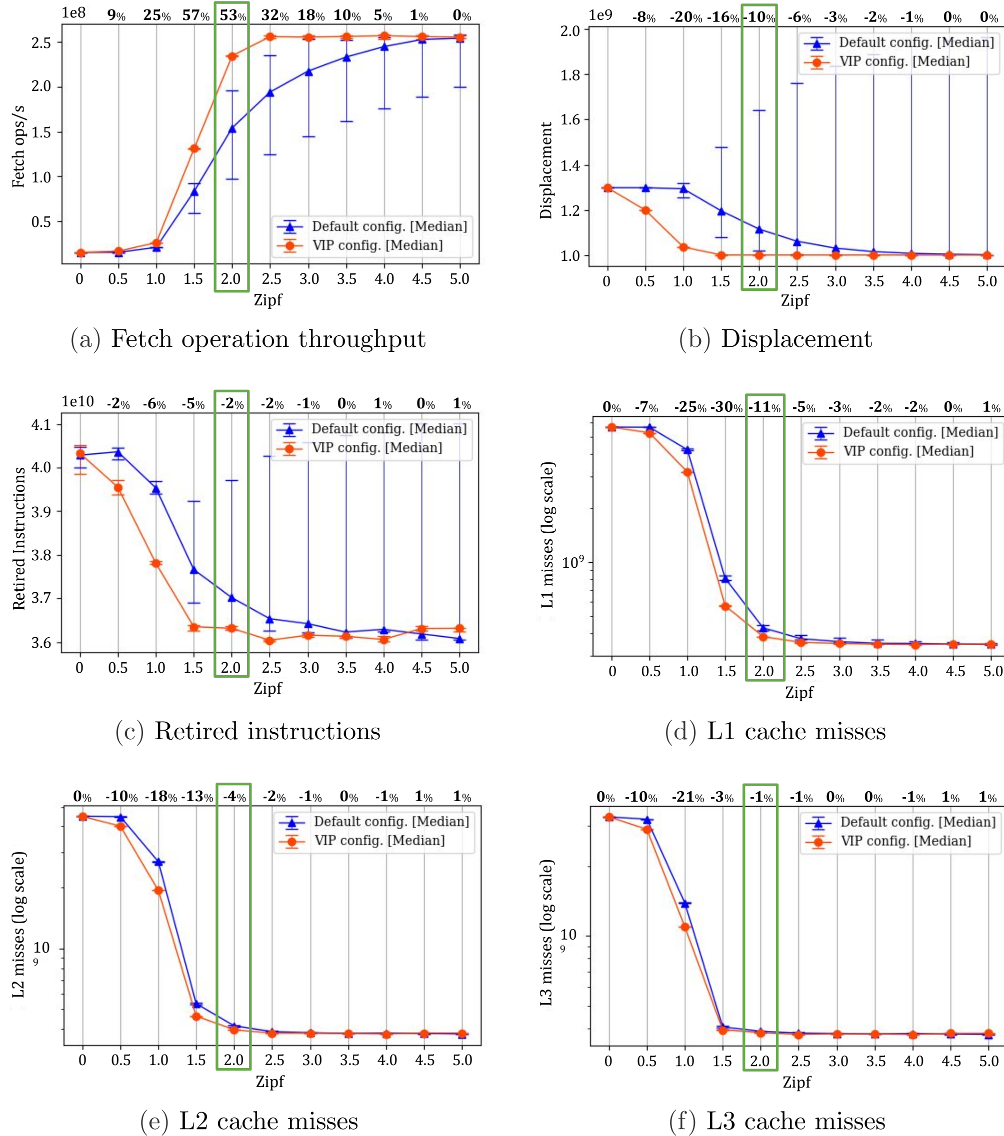


(a) Default configuration. A total displacement of 12 (=2×(2+1+3)) is required to process the fetch requests. The less popular keys in the path of popular keys need to accessed as well.



(b) VIP configuration. A total displacement of 6 (=2×(1+1+1)) is required to process the fetch requests. Only the popular keys are accessed.

Figure 3.3: Processing fetch requests in the Default vs the VIP configuration. Unpopular keys have been grayed out. The total *displacement* (number of keys accessed) is higher in the Default configuration requiring more pointer dereferences. Also, the effective hot set is larger, increasing the likelihood of cache misses relative to the VIP configuration.



(a) Fetch operation throughput

(b) Displacement

(c) Retired instructions

(d) L1 cache misses

(e) L2 cache misses

(f) L3 cache misses

Figure 3.4: Relative performance of the VIP vs the Default configurations as the skew in popularity increases. One billion fetch requests are issued to a hash table with 10M keys (load factor 0.6) for varying levels of skew from $zipf = 0$ to $zipf = 5$. Each reported data point is the median over 10 runs with different random seeds. Percentages indicated at the top of each plot is the difference between median metrics of the VIP vs the Default configuration. The gain in fetch operation throughput varies with skew, and we obtain 53% increase in throughput for medium skew ($zipf = 2.0$). Lesser number of cache misses and instructions executed contribute to the gain obtained from the VIP configuration. Further observations are discussed in §3.4.2.2.

the bucket chain (§3.2.1). Thus, when inserting keys in the sorted order, entries are automatically placed in decreasing order of popularity as more popular keys are inserted later and are ahead in the bucket chain. The Default configuration is generated using the default parameters of Wiscer.

3.4.2 Impact of Increasing Skew

3.4.2.1 Workload

We compare the throughput of fetch operations in the Default and VIP configurations. We use Wiscer (Table 3.1) to generate fetch requests with increasing levels of skew ($zipf = 0$ to 5 in steps of 0.5) which are issued to a hash table with 10 million keys at a load factor of 0.6 ($= 10^7/2^{24}$). For each level of skew and hash table configuration, Wiscer is run with 10 distinct random seed values to populate the hash table and generate the workload. Each random seed results in a different arrangement of keys in the hash table. The popularity distribution is static, i.e., the rank of the keys remains the same throughout a run. One billion fetch requests are issued to the hash table for each random seed, and the data points reported in Fig. 3.4 are the median statistics over the 10 runs. We have run experiments on smaller (1M entries) and larger (100M entries) hash tables and found the trends to be similar.

3.4.2.2 Results

The results of this experiment are shown in Fig. 3.4. The gain in throughput ranges from 9%-57% depending upon the level of skew in popularity. Below we discuss our takeaways from the performance metrics measured using Wiscer:

- **Throughput**: The gap in performance between the VIP and the Default configuration increases up to $zipf = 2$ (medium skew), and gradually diminishes as the skew becomes very high ($zipf =$ 4.5 or 5). This behavior is correlated with the hot sets becoming smaller as the skew increases and becoming (L1/2/3) cache resident at different rates for the two configurations.
- **Displacement**: As expected, the displacement encountered in the VIP configuration is lower than the Default (see Fig. 3.3). For $zipf = 1.5$ and up, the total displacement becomes close to 1B (for 1B fetch requests), indicating that popular keys are at the front of their chains (displacement = 1) in the VIP configuration. For the Default configuration, the median displacement approaches 1B at higher levels of skew ($zipf \geq 4$), but the variance is high as some random seeds can result in the popular keys placed further in the chains (however the likelihood of this happening is low as the load factor is not very high).

- **Instructions Executed**: The instructions executed are lower in the VIP configuration (up to 6% lower in the best case). The relative trend observed is similar to that of displacement, as the number of instructions executed is correlated with the number of keys accessed.
- **Cache misses**: The VIP configuration becomes L3 and L1 cache resident (at $zipf = 2$ and 2.5 respectively) more quickly compared to the Default configuration (at $zipf = 3.0$ and 4.5 respectively), which is expected as the hot set of the former is smaller than the latter (Fig. 3.3). At very high skew ($zipf = 4.5$ and 5), both the configurations are L1 resident and correspondingly, we do not observe much difference in the throughput. This indicates that caching has a big impact on the performance of hash tables.

Overall, we note that since the hot set of the VIP configuration is smaller than the Default, we encounter lower cache misses at all levels of cache. This contributes to the gain in performance we obtain from the VIP configuration.

Another important observation we make is that the metric *displacement* indicates the goodness of the hash table configuration. The VIP configuration has lower displacement than the Default in all cases (the VIP configuration has the lowest possible displacement for a given data set, hash table size, hash function, and request skew; see §3.5.2.3). We use this metric in building the mechanisms for sensing and dynamically switching-on/off learning (§3.5.2.3).

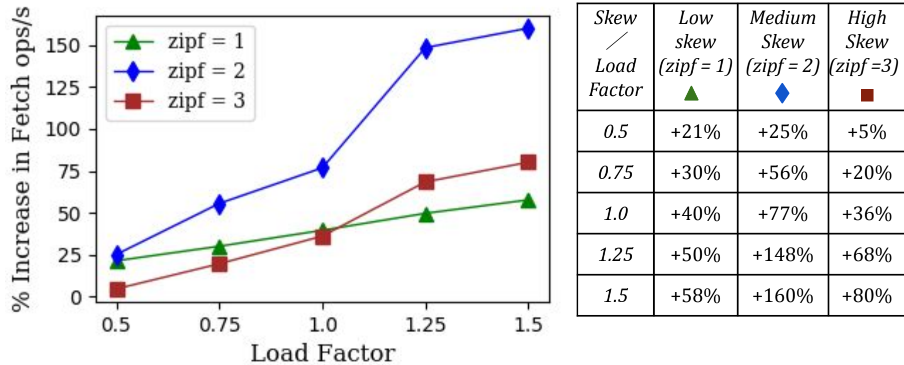
3.4.3 Impact of Increasing the Load Factor

3.4.3.1 Workload

In this experiment, we increase the load factor while holding the size of the hash table constant. Similar to §3.4.2.1, we run one billion fetch operations on a hash table with 2^{24} buckets while varying the load factor from 0.5 to 1.5 in steps of 0.25 (this is achieved by increasing *initialSize* from 2^{23} to $3 \cdot 2^{23}$). Each configuration is run with 10 distinct random seeds and we compare the median statistics over the 10 runs.

3.4.3.2 Results

Fig. 3.5 shows the median gain obtained as we increase the load factor – we obtain 1.6x, 2.6x, and 1.8x higher throughput from the VIP configuration at low ($zipf = 1$), medium ($zipf = 2$), and high skew ($zipf = 3$) respectively at load factor 1.5. In all cases, the gain from the VIP configuration increases as the load factor increases, which is expected as the likelihood of collisions is higher when more keys are present in the hash table. We find that the performance metrics of the VIP configuration are mostly stable (refer to Table 3.2) indicating a stable hot set size, while



Skew ⁄ Load Factor	Low skew (zipf = 1) ▲	Medium Skew (zipf = 2) ♦	High Skew (zipf =3) ■
0.5	+21%	+25%	+5%
0.75	+30%	+56%	+20%
1.0	+40%	+77%	+36%
1.25	+50%	+148%	+68%
1.5	+58%	+160%	+80%

Figure 3.5: Roofline gain in throughput from the VIP vs the Default configuration as the load factor increases. Keeping the number of buckets fixed at 2^{24}, we increase the load factor from 0.5 to 1.5. The performance gain obtained from the VIP configuration increases with the load factor, and can be as high as 160% (2.6x) for medium skew at load factor 1.5.

the performance of the Default configuration becomes steadily worse as the effective hot set grows larger with the load factor.

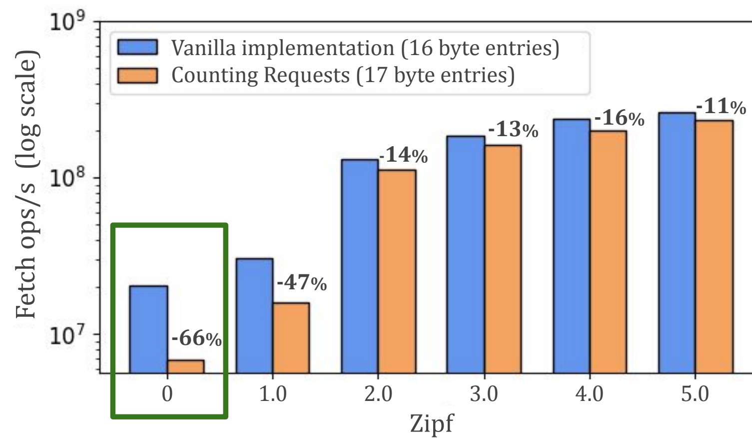
3.5 Adapting to Popularity on-the-fly

In this section, we first highlight the challenges of learning in-the-loop (§3.5.1) which motivated the lightweight mechanisms we built for VIP hashing. We then describe how we learn, adapt, sense, and dynamically control the overhead on the fly (§3.5.2-3.5.3).

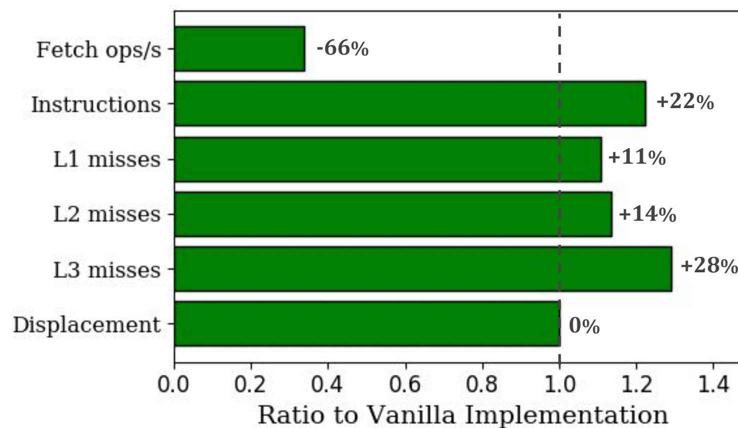
3.5.1 Learning In-the-Loop is Costly

Hash tables execute a tight loop of instructions – compute the hash function, access keys in the bucket, and perform required operations to process the request. Adding any amount of additional computation or storage to this loop can degrade performance considerably. To demonstrate this behavior, we conduct a simple experiment of adding a 1-byte requests counter per key in the hash table, such that the entries become 17 bytes long (8 byte key and value, and 1 byte counter).

We use Wiscer to compare the performance of the vanilla implementation of hash table (16 byte entries) to the implementation with request counters (17 byte entries). We issue 500M fetch requests to a hash table with 1M entries (load factor $0.95 = 10^6/2^{20}$) for different levels of skew in the popularity distribution ($zipf$ = 0 to 5 in steps of 1). The remaining configuration options of Wiscer are set to the defaults (refer to Table 3.1). Fig. 3.6 shows the relative performance of the



(a) Loss in performance when adding a 1-byte counter per key in the hash table. Both hash tables are identical (in Default configuration) except for the size of the entries (16 vs 17 bytes).



(b) Relative metrics for zipf = 0. Instructions executed and cache misses increase after adding the 1-byte counter.

Figure 3.6: The effect of adding a 1-byte requests counter per key in the hash table. 500M fetch operations are issued to a hash table with 1M keys at load factor 0.95. Performance can take a significant hit – we observe a 66% loss in fetch operation throughput at $zipf = 0$. This experiment demonstrates the sensitivity of hash tables to effects of caching and computation, which makes learning on the fly challenging.

Table 3.2: Relative Metrics of VIP vs Default configuration as we increase the load factor (*lf*) at $zipf = 2$. The trends for low and high skew are similar.

lf	**Throughput (fetch ops/s)**	**Avg. Disp- -lacement**	**L3 Misses**	**L1 Misses**
0.5	235M *vs* 188M *(+25%)*	1.0 *vs* 1.03 *(-3%)*	378M *vs* 385M *(-1.8%)*	380M *vs* 412M *(-8%)*
1	236M *vs* 134M *(+77%)*	1.0 *vs* 1.17 *(-15%)*	376M *vs* 387M *(-2.6%)*	380M *vs* 436M *(-13%)*
1.5	236M *vs* 90M *(+160%)*	1.0 *vs* 1.62 *(-38%)*	382M *vs* 392M *(-2.6%)*	382M *vs* 458M *(-17%)*

two hash table implementations at different levels of skew in the workload. There is a significant loss in throughput ranging from 11-66% due to increase in cache misses and instructions executed.

Counting requests is a fundamental requirement for learning the popularity distribution. However, this experiment shows that even adding a small amount of additional memory can hurt performance significantly. Thus, the challenge here is to work with a restricted "budget" when learning in-the-loop, to balance the gains against the overhead of learning.

3.5.2 VIP Hashing

From §3.5.1, we know that using additional memory and computation can really hurt the performance of hash tables. In this section, we describe how VIP hashing overcomes these challenges by using lightweight mechanisms for learning and adapting to the popularity distribution (§3.5.2.2), while controlling the overhead by sensing and dynamically switching-on/off learning as necessary (§3.5.2.3). We first give an overview of VIP hashing (§3.5.2.1) followed by describing the mechanisms used in detail (§3.5.2.2-3).

3.5.2.1 Overview

Fig. 3.7 shows the VIP hashing method. At any given time, there are three possible modes that the hash table implementation can be in – *learn+adapt*, *sense*, and *default* (or vanilla). In the learn+adapt mode, the hash table learns the popularity distribution and rearranges keys to move closer to the VIP configuration. This mode is costly in terms of both computation and storage, and we control how much we run this mode by configuring the parameter N_L. The learn+adapt mode

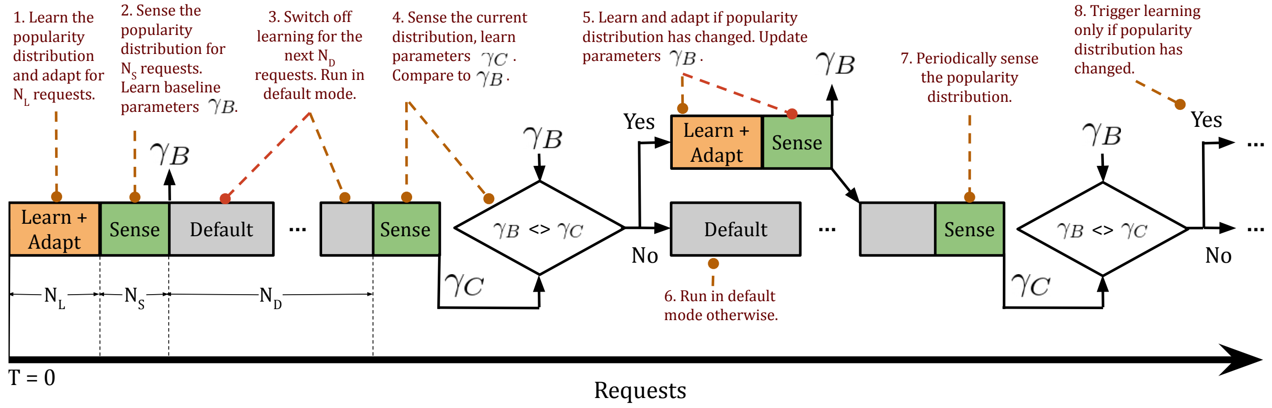


Figure 3.7: Overview of VIP Hashing. At any time, the hash table is in one of the three modes – *learn+adapt*, *sense*, or *default*. The amount of time spent on learn+adapt mode is controlled through the parameter N_L to cap the overhead of executing on the fly. The popularity distribution is sensed periodically and learning is triggered only when a change is detected.

is run at the start, and subsequent triggers of this mode happen only if the popularity distribution changes, which is determined during the sense mode.

The sense mode is triggered after the learn+adapt mode to measure some statistics (γ_B) that characterize the popularity distribution. These statistics require a total of 24 bytes of memory for the whole hash table (irrespective of the size) and a few additional arithmetic operations in the loop. Since the memory and computation footprint of this mode is low, it does not add much overhead to the execution. The sense mode is run for N_S requests at a time, and is triggered periodically (every N_D requests) to characterize the popularity distribution at the time (γ_C). Comparing the statistics (γ_B and γ_C) helps determine if the popularity distribution has changed, and informs the decision of whether to switch on learning.

The default mode is the vanilla implementation of chained hashing (§3.2.1) with 16 byte entries. There is no additional overhead of storage or computation. This mode is run most of the time ($N_D > N_L$, N_S), so the performance is close to the vanilla implementation of hash table in the worst case.

In the following sections, we discuss the mechanisms we use for the learn+adapt (§3.5.2.2) and sense (§3.5.2.3) modes. We discuss our choice of parameters (N_L, N_S, N_D, etc.) in §3.5.3, that allow us to balance the performance gains against the overhead of learning.

Algorithm 3 Learning and Adapting on-the-fly

```
1: procedure FetchAdaptive(requests)
2:     ht ← getHashTable()
3:     /* Requests are counted in a separate data structure*/
4:     req_cnt_ht ← getRequestsCountingHashTable()
5:     for r in requests do
6:         hash ← murmurHash(r.key)
7:         ht_entry ← ht[hash]
8:         req_entry ← req_cnt_ht[hash]
9:         /* Keep track of entry with minimum requests */
10:        min_req_ht_entry = ht_entry
11:        min_req_entry = req_entry
12:        while ht_entry and ht_entry.key ≠ r.key do
13:            if req_entry.count < min_req_entry.count then
14:                min_req_ht_entry = ht_entry
15:                min_req_entry = req_entry
16:            ht_entry = ht_entry.next()
17:            req_entry = req_entry.next()
18:        if ht_entry == null then
19:            r.found = false
20:            continue
21:        r.found = true
22:        r.value = ht_entry.value
23:        req_entry.count = req_entry.count + 1
24:        if req_entry.count > min_req_entry.count then
25:            /* Swap this entry with the min requests entry */
26:            swap(ht_entry, min_req_ht_entry)
27:            swap(req_entry, min_req_entry)
28:    /* Reclaim cache space by clearing req_cnt_ht */
29:    clearCache(req_cnt_ht)
```

3.5.2.2 Learning & Adapting

Algorithm 3 describes how we learn the popularity distribution and adapt to the skew on the fly. The popularity of a key is estimated as the proportion of requests made to the key (§3.2.2). Thus, learning the popularity distribution requires counting requests, which we know is challenging from the experiments in §3.5.1.

To overcome the challenge of counting requests in-the-loop, we perform two optimizations. First, we count requests in a separate data structure that mimics the hash table in arrangement (for every entry in the hash table, there is a corresponding entry in the request counting hash table). Although this temporarily requires more memory (about 50-60% increase in memory usage depending on the load factor) than maintaining a counter per key in the hash table, the cost is incurred only during the learn+adapt mode. Second, at the end of the learn+adapt mode, we clear the requests counting hash table (*req_cnt_ht*) from the cache by issuing cache flush instructions (`_mm_clflushopt` on Intel CPUs [inta]), which mitigates the cache pollution caused by the requests counting data structure used during the learn+adapt mode.

To attain the VIP configuration, we need to sort the keys in descending order of popularity in the bucket chains. Given that the proportion of requests made to a key is an estimate of popularity, we use Algorithm 3 to stochastically sort the keys in descending order of requests received on the fly. When performing a fetch operation, we keep track of the entry with minimum requests (*min_req_ht_entry*) encountered in the path to the entry being fetched. If the entry being fetched has received more requests, then it is swapped with the *min_req_ht_entry* and it moves forward in the chain. We propose the following theorem:

Theorem 1 : Let there be a bucket chain with n keys K_1, K_2 ... K_n which have popularity $p_1 > p_2 ... > p_n > 0$. Let the keys be in a random order in the chain. Then, by applying Algorithm 3, the keys will converge to the sorted order of popularity as number of fetch requests $N \to \infty$.

Proof: From the frequentist definition of probability, we can be sure that a more popular key will receive more requests compared to a less popular key as $N \to \infty$. This will hold pairwise for all the keys $K_1, K_2, ..., K_n$ in the bucket, which motivates the following lemma.

Lemma 1 Let $\{K_i\}$ be keys in a bucket with probability $\{p_i\}$, $i \in [N]$. Let K_1 be the most popular key in the bucket, i.e., $p_1 > p_j \ \forall j \in \{2, .., N\}$. Let the initial order of keys be random. Then, by running Algorithm 1, K_1 will be at the front of the chain as $N \to \infty$.

Proof: Suppose K_1 is at displacement $d > 1$ and has received n requests. Let there be keys $K'_1,\ ..,\ K'_{d-1}$ in front of K_1 that have received requests $n_1,\ ..,\ n_{d-1}$ respectively. From the frequentist definition of probability, we have

$$\lim_{N\to\infty} n > n_i,\ \forall\ i \in [(d-1)]$$

K_1 would have received more requests than all the keys in front of it as $N \to \infty$. From Algorithm 1, on the last request that K_1 received, it should have been swapped with a key with lower number of requests ahead of it. This contradicts our assumption that K_1 is at position $d > 1$. $\square$

Thus, the most popular key in the chain will be in the front as number of requests approaches infinity. By recursively applying Lemma 1 to the remaining keys in the bucket, we can prove that the keys will be in the sorted order of popularity as $N \to \infty$. $\square$

There are two noteworthy properties of Algorithm 3. First, the VIPs move to the front quickly, as they can skip over multiple entries in a single fetch request. This algorithm is, in essence, similar to selection sort as we are moving the entry with minimum requests to the end of the (sub-)chain being accessed. An alternative would be to compare only adjacent keys (bubble sort), which empirically requires more requests for a VIP to move to the front.

Second, the cost of swapping is amortized, as there is at most one swap performed per fetch operation. This approach is faster compared to performing a full sort on every request, or sorting at the end after counting requests for some time (we will have to access all the buckets in order to perform a full sort, which will block operation, incur cache misses, and pollute the cache).

3.5.2.3 Sensing & Dynamically Switch-on/off Learning

Algorithm 4 describes how we sense some key statistics of the popularity distribution, which enable us to dynamically switch-on learning only when the distribution has changed (Algorithm 5). While there are multiple ways to quantify the difference between two probability mass functions (*pmf*s) [Wikc, Wikh, Wikb], we choose a lightweight statistic to compare distributions – *average displacement*. In §3.4.2.2, we saw that displacement encountered indicates the "goodness" of the hash table configuration. Every popularity distribution imposes a pmf over the displacement encountered on a request, which is a derived random variable. Formally stated:

Algorithm 4 Sensing

```
1:  procedure FetchSensing(requests)
2:      ht ← getHashTable()
3:      /* Metrics to track */
4:      disp ← 0                                        ▷ cumulative displacement
5:      disp_sq ← 0                                     ▷ cumulative disp. square
6:      count ← 0                                       ▷ number of requests
7:      c = 0.95                                        ▷ confidence level of the interval
8:      for r in requests do
9:          hash ← murmurHash(r.key)
10:         ht_entry ← ht[hash]
11:         d ← 1
12:         while ht_entry and ht_entry→ key ≠ r.key do
13:             ht_entry =  ht_entry.next()
14:             d = d + 1
15:         if ht_entry == null then
16:             r.found = false
17:             continue
18:         r.found = true
19:         r.value = ht_entry.value
20:         count = count + 1
21:         disp = disp + d
22:         disp_sq = disp_sq + d × d
23:     /* Estimating mean u, variance v, and C.I. width w*/
24:     u = disp/count
25:     v = disp_sq/(count − 1) − disp^2/(count ∗ (count − 1))
26:     w = sqrt(−2.v.log(1 − c)/count)                 ▷ Gaussian tail bound
27:     γ = (u, w)
28:     return γ
```

Algorithm 5 Dynamically Switch-on/off Learning

1: **procedure** HASDISTRIBUTIONCHANGED(γ_B, γ_C)
2: $(u_B, w_B) = \gamma_B$
3: $(u_C, w_C) = \gamma_C$
4: **if** $|u_B - u_C| > (w_B + w_C)$ **then**
5: **return** true
6: **else**
7: **return** false

Axiom 1 Let $K_1,\ K_2,\ ...,\ K_N$ be N keys in the hash table with popularity $p_1,\ p_2,\ ..., p_N$ ($\sum p_i = 1$) at displacement $d_1,\ d_2,\ ...,\ d_N$ ($d_i \leq N$). Let D be the random variable of the displacement encountered on a successful fetch request. Then,

$$P(D = d) = \sum_{i=1}^{N} p_i \cdot 1_{d_i = d}$$

i.e, the probability that displacement d is encountered on a fetch request is the probability that any of the keys with displacement d were fetched. The average displacement is calculated as

$$\mu_D = E[D] = \sum_{i=1}^{N} i \cdot P(D = i)$$

We make the following observation:

Axiom 2 The VIP configuration minimizes E[D] over all possible arrangements of keys in the hash table for a fixed load factor, popularity distribution, and hash function.

The VIP configuration orders keys by popularity, thus giving more "weight" to lower values of D which minimizes the average displacement. It is straightforward to see that for a given hash table configuration, two popularity distributions with different average displacement will not be identical (although the opposite is not true). Thus, a change in average displacement reflects a shift in the popularity distribution.

The parameters we learn from sensing are $\gamma = (\hat{\mu}_D, \hat{w}_D) = (u, w)$ (Algorithm 4), where $\hat{\mu}_D$ is the estimated average displacement, and $\hat{w}_D$ is the width of the confidence interval around $\hat{\mu}_D$ obtained using Gaussian tail bounds (§3.2.2). Average displacement is estimated as

$$\hat{\mu}_D = \frac{\sum_{i=1}^{N_S} D_i}{N_S}$$

which is the sample mean* of displacement encountered D_i ($1 \leq i \leq N_S$) over N_S fetch requests in the sense mode. Similarly, we also estimate sample variance $\hat{\sigma}_D^2$ (§3.2.2).

We further characterize the pmf by building a confidence interval using Gaussian tail bounds (§3.2.2). The width ($\hat{w}_D$) of the interval at confidence level c ($c = 0.95$ in our experiments) is calculated as

$$\hat{w}_D = \sqrt{\frac{-2 \cdot \hat{\sigma}_D^2 \cdot (1 - c)}{N_S}}$$

Note that $\hat{\sigma}_D$ is estimated variance from a sample of N_S observations, and $(\hat{\mu}_D - \mu_D)$ only approximately Gaussian according to CLT (§3.2.2). Thus, the width $\hat{w}_D$ obtained by applying Gaussian tail bounds is a heuristic.

We switch-on learning (Algorithm 5) only if we detect a significant change in the average displacement. Given two sets of parameters $\gamma_B = (u_B, w_B)$ and $\gamma_C = (u_C, w_C)$ where u_B and u_C are estimated means, we check if the confidence intervals are disjoint. If so, then heuristically with a probability $c^2 = (0.95)^2 = 0.9$, we can be sure that the real means are not equal and the distributions have diverged. Thus, we detect changes in popularity distribution in a non-intrusive manner by computing lightweight statistics.

3.5.3 Parameters

The parameters $\mathrm{N_L}$, $\mathrm{N_S}$, and $\mathrm{N_D}$ determine how long the hash table runs in learn+adapt, sense, and default modes respectively. Our goal is to choose these parameters such that the gains of learning are balanced against the overhead.

Our choice of parameters is general, made using theoretical and empirical evidence that is independent of the popularity distribution. Thus, our techniques (§3.5.2) apply to any distribution with skew irrespective of its specific properties. Note that it is possible to further tune the parameters and the techniques with additional knowledge such as total number of requests, patterns in the workloads, family of distribution, etc.

3.5.3.1 Allocating the budget for learning – $\mathbf{N_L}$ *vs* $\mathbf{N_D}$

Learning in-the-loop is costly. In our experiments, we find that the learn+adapt mode can be as much as $4x$ slower than the vanilla implementation in the worst case (under no skew for different hash table sizes from 1M to 100M keys). If a total of $(N_L + N_D)$ requests are issued, the loss in

*Note that instead of sampling, we could also use the request counting data structure (*req_cnt_ht* in §3.5.2.2). However, this would incur cache misses and also pollute the cache affecting performance (§3.5.1).

throughput due to the learn+adapt mode would be:

$$1 - \frac{T_{vanilla}}{T_{vip}} \leq \left(1 - \frac{N_D.t + N_L.t}{N_D.t + N_L.4.t}\right)$$

assuming that the vanilla implementation takes time t on an average to process each request. We cap the overhead of learning to at most 5% by choosing $N_{\mathrm{D}} = 60 \cdot N_{\mathrm{L}}$ in our experiments (i.e, learn+adapt mode is run for at most $\frac{1}{61}$ of the total requests). More generally, the cap on overhead is $(1 - \frac{61}{(60+k)})$, where k depends on the experimental configuration ($k = 4$ on our hardware). Thus, we cap the overhead of learning by fixing a budget for $\frac{\mathrm{N_L}}{\mathrm{N_D}}$.

3.5.3.2 Choosing $N_{\mathbf{L}}$ – how much to learn?

The learn+adapt mode is run for N_L requests at a time. Our goal is to capture the popularity distribution while learning for a finite number of requests. From previous work [Can20], we know that it takes $\Theta(N)$ i.i.d. samples to learn a probability mass function over N items (with error $\epsilon = 1$ in KL divergence compared to the true pmf). When the cardinality of the hash table is not known/can vary, we choose $N_L = 1.5 \cdot (htsize)$, i.e, 1.5 times the number of buckets in the hash table. Since we maintain a load factor of at most 1.5 at all times, the number of keys in the hash table $N \leq 1.5 \cdot htsize$, which satisfies our requirements.

3.5.3.3 Parameters for sensing – $N_{\mathbf{s}}$ and c

We sense the distribution for $\mathrm{N_S}$ requests at a time to estimate the average displacement $\hat{\mu}_D$ and build an interval with confidence c. Since the load factor is low and the longest chain length is likely to be low as well (except in pathological cases), we have found that choosing N_S to be a large number (1000) has been sufficient in our experiments. We build a $c = 95\%$ confidence interval that gives us a heuristic probability of $c^2 = (0.95)^2 = 0.9$ when we detect a shift in the popularity distribution. By increasing (decreasing) the confidence level, we can be less (more) sensitive to changes in popularity.

3.6 Applications

3.6.1 PK-FK Hash Joins

Hash tables are frequently used in database systems for processing join queries. In this section, we describe how VIP hashing can improve the performance of primary key-foreign key (PK-FK) hash joins in the presence of skew.

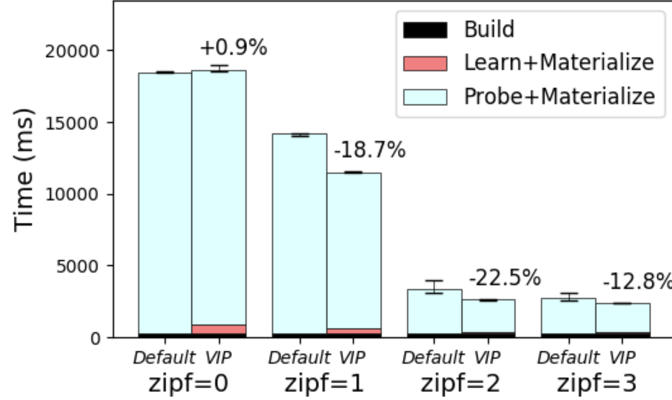


Figure 3.8: Performance of PK-FK canonical hash join on tables R and S ($|R| : |S| = 1 : 16$) using the default and VIP hash table implementations. For medium skew, we observe a 22.5% reduction in median (over 10 random seeds) total execution time.

3.6.1.1 Experimental Setup

Motivated by past research [BTAO13, BLP11, KTMO09], we consider the canonical PK-FK join query on tables R and S ($|R| \leq |S|$) with 8-byte integer attributes (16-byte tuples). Skew can arise in PK-FK relations [BLP11, BTAO13] when some keys occur more frequently than others in the outer relation S. We use Wiscer to instantiate R and S using the sequential key pattern for primary keys in R, and varying the level of skew in the outer relation S from uniform ($zipf = 0$) to high ($zipf = 3$) for 10 distinct random seeds. We compare the performance of the canonical hash join algorithm [BTAO13, KTMO09] implemented using the default and VIP hash tables, while materializing pointers to output tuple pairs. We assume that the tuples in S are i.i.d, i.e, the popularity distribution is static. We explore effects of dynamic popularity distribution in §3.6.2.

3.6.1.2 Default vs VIP Hash Join

Fig. 3.8 shows the relative execution time of the default vs VIP hash join implementations. The cardinalities of R and S are 12M and 192M respectively ($|R| : |S| = 1 : 16$) [BLP11, BTAO13], and the load factor is 1.4 ($= 12 \cdot 10^6/2^{23}$). For medium skew in the outer relation, the average displacement encountered by the default hash join implementation is 1.23 (Table 3.3)[†].

For the case of canonical hash join query, the learning budget of the VIP hash table implementation can be calculated in advance while maintaining $N_L : N_D = 1 : 60$ (§3.5.3) since we almost

[†]Note that the average displacement is low for the default configuration in this case, since the keys are sequential. Holding the load factor constant, randomly generated keys result in a median (over 10 random seeds) average displacement of 1.48.

Table 3.3: Relative metrics for default and VIP hash join at $zipf = 2$, $|R| : |S| = 1 : 16$.

Metric	*Default*	*VIP*	*Diff*
Time	3.4s	2.6s	-22.5%
Avg. Displacement	1.23	1.0003	-18.7%
L3 Misses	75.5M	75.3M	-0.3%
L2 Misses	127.9M	124.6M	-2.6%
L1 Misses	161.2M	155.7M	-3.4%
Instructions	8.5B	8.2B	-3.5%

always know the cardinalities of the relations from system catalogs. Learning is triggered at the beginning of the probe phase with a budget of $N_L = min(|R|, \frac{|S|}{61}) = \frac{16 \cdot |R|}{61} = 0.26 \cdot |R|$ lookups from the outer relation. Learning takes about 3% of the total execution time, ranging from 70-600ms depending on the level of skew. Note that the average displacement of the VIP hash join implementation is very close to 1 (Table 3.3) indicating that the learning mechanism efficiently captures the popularity distribution, and reduces cache misses and instructions executed.

To show the impact of varying the learning budget, we repeated the experiment for lower and higher cardinality ratios. For a ratio of $1 : 4$, we have a learning budget of $\frac{4 \cdot |R|}{61} = 0.07 \cdot |R|$ requests and the overall reduction in execution time is 18.6%. On the other hand, a cardinality ratio of $1 : 64$ allows a learning budget of $|R| = min(|R|, \frac{64 \cdot |R|}{61})$ and results in 25.8% reduction in execution time. Thus, the available learning budget impacts the gain in performance.

3.6.1.3 Application to Skewed TPC-H

We focus our attention on TPC-H query 9 [tpc], which is the most expensive TPC-H query involving multiple PK-FK joins. We implemented VIP hashing in DuckDB [RM19], an in-memory vectorized DBMS, to speed up PK-FK hash joins in single-threaded mode. Fig. 3.9 shows the median execution time of VIP hash join relative to the default, tested on skewed TPC-H data [KPKP22b] at varying levels of skew for 10 different random seeds. VIP hash join reduces the end-to-end query execution time by 20% at $zipf = 1$ and $zipf = 1.5$, while the increase in execution time at lower skew is negligible. The remaining TPC-H queries spend $< 1\%$ of the total execution time in skewed PK-FK hash joins, and consequently the impact of VIP hashing is negligible.

3.6.2 Point Queries

Another common use of hash tables is for in-memory indexing in database systems [Frø, mar13] and in key-value stores [redb, He] for processing point queries. In this section, we evaluate VIP

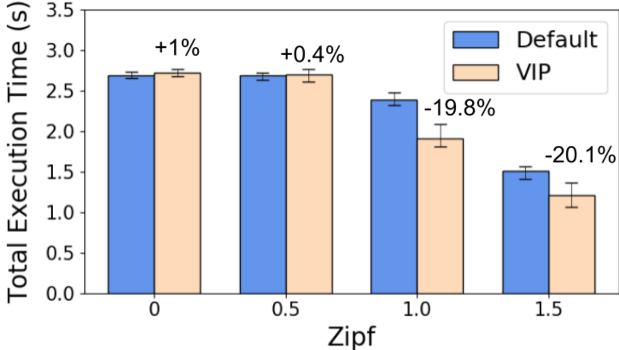


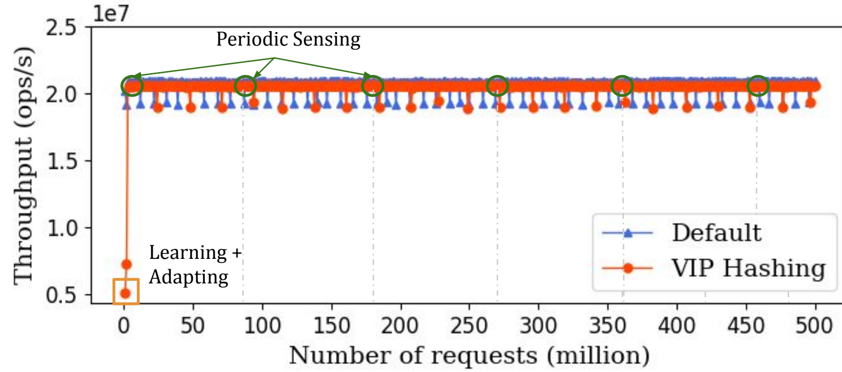
Figure 3.9: Execution time of TPC-H query 9 (scale factor = 1) on DuckDB. VIP hashing speeds up PK-FK hash join probes, and results in 20% reduction in median (over 10 random seeds) end-to-end query execution time at $zipf = 1$ and $zipf = 1.5$.

hashing against a range of workloads generated using Wiscer that highlight the robustness of our techniques for learning in-the-loop under different conditions. In all the experiments, we assume no prior knowledge of the characteristics of the request distribution. The first two workloads (§3.6.2.1-§3.6.2.2) involve fetch operations, and the last two (§3.6.2.3-§3.6.2.4) perform insert and delete operations.

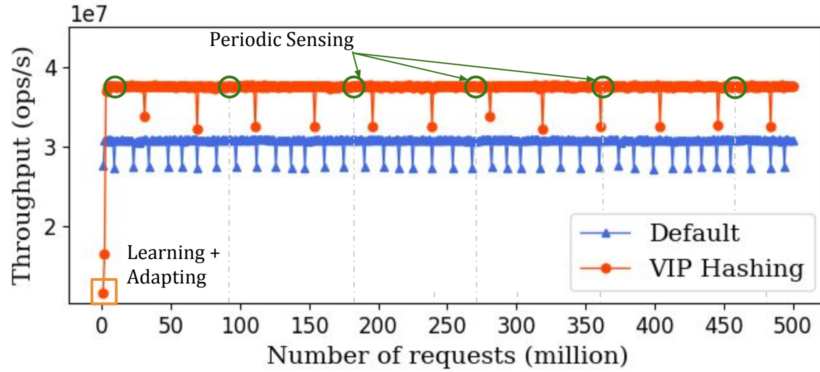
We run these workloads on a hash table with 1M entries (load factor $0.95 = 10^6/2^{20}$) in the Default configuration. Each of these workloads issue 500M operations to the hash table at low skew ($zipf = 1$) unless specified otherwise. The performance gain under medium skew ($zipf = 1.5$) is higher, and those results are included in [KPKP22b]. The remaining configuration options of Wiscer are set to the defaults (Table 3.1). We compare the performance of VIP hashing to the default hash table in Fig. 3.10-3.12.

3.6.2.1 Static Popularity

In this workload, the popularity of keys in the hash table remains the same throughout. For the case of uniform popularity distribution ($zipf = 0$), the loss in throughput is 2% (Fig. 3.10a) which is within our budget of 5% (§3.5.3.1), whereas for low skew ($zipf = 1$), we obtain a net gain of 22% (Fig. 3.10b). Since the popularity distribution is static, the learn+adapt mode is triggered only at the start of the experiment for $1.5 \cdot htsize$ requests. The periodic runs of the sense mode do not detect a change in popularity and the learn+adapt mode is not triggered again, thus minimizing the overhead of learning.

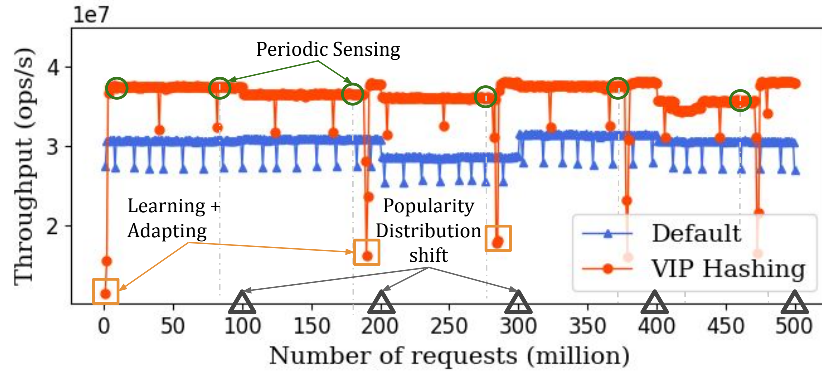


(a) Static popularity (§3.6.2.1) with $zipf = 0$ (uniform distribution). Since there is no skew in popularity, no performance gain can be obtained from VIP hashing. Learning adds overhead to VIP hashing ($4x$ slower), and is only triggered at the start for $(1.5 \cdot 2^{20})$ requests (0.3s). Subsequent sensing of the popularity distribution does not detect any change, and learning is not triggered. Total loss in throughput is 1.9%, which is within our allocated budget.

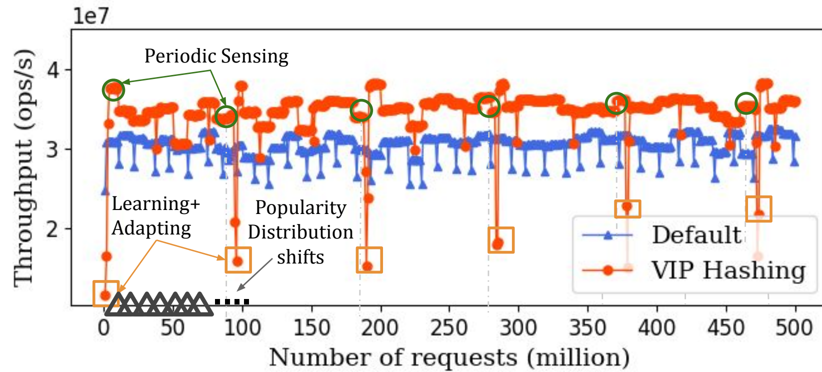


(b) Static popularity (§3.6.2.1) with $zipf = 1$ (low skew). Learning is only triggered at the start and is 3x slower than the default (0.13s vs 0.05s respectively). Sensing does not detect any changes to the popularity distribution, so learning is not triggered again. The overhead of learning is offset by the gain in performance from the VIP configuration. We observe an overall increase in throughput of 21.8%.

Figure 3.10: Comparing the performance of VIP hashing to the default (vanilla) implementation of hash table when subjected to identical fetch-only workloads with static popularity distribution. Workload 3.10a has uniform popularity distribution ($zipf = 0$) and workload 3.10b is run with low skew ($zipf = 1$).

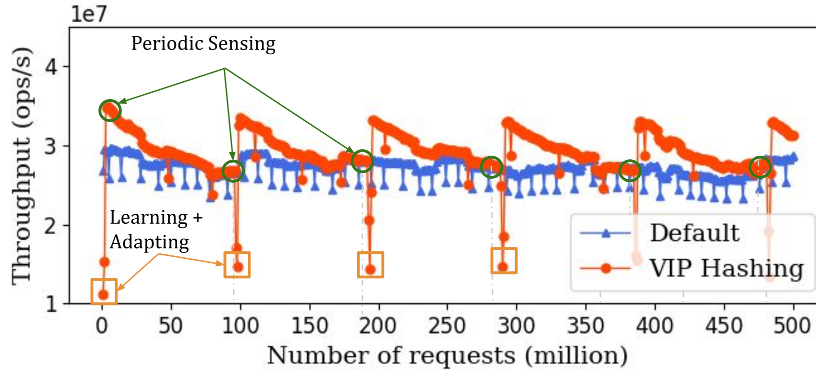


(a) Medium churn rate (§3.6.2.2) with $zipf = 1$. Popularity distribution shifts every 100M requests by 25% (top 21 out of 1M keys are replaced by less popular keys). Distribution shift increases average displacement and can reduce performance (notice drop in performance of VIP hashing at 200M requests). Sensing triggers learning whenever it detects a significant increase in average displacement. Throughput increases by 18.9% overall.

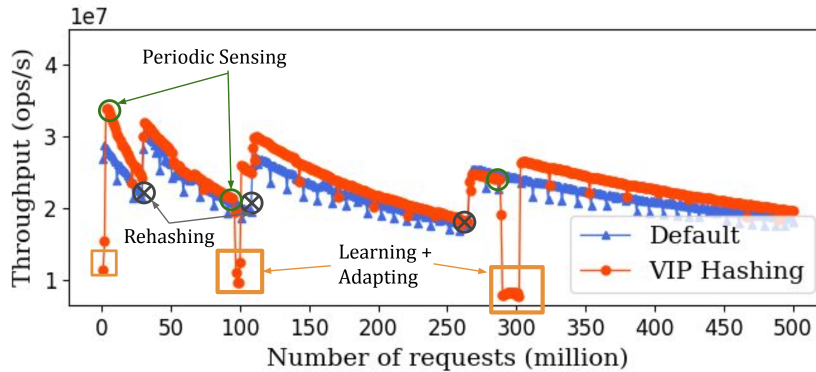


(b) High churn rate (§3.6.2.2) with $zipf = 1$. Popularity distribution shifts every 10M requests by 50% (top 750 out of 1M keys are replaced by less popular keys). The benefit of learning dimishes as the popularity order becomes shuffled. Periodic sensing triggers learning every time, as frequent distribution shifts cause significant change in average displacement. Overall, 11.8% increase in throughput is observed.

Figure 3.11: Comparing the performance of VIP hashing to the default (vanilla) implementation of hash table when subjected to identical fetch-only workloads with dynamic popularity distribution at low skew ($zipf = 1$). In workloads Fig. 3.11a and 3.11b, popularity distribution changes at a medium and high rate respectively.



(a) Steady state (§3.6.2.3) with $zipf = 1$. 98% fetch requests, 1% insert requests, and 1% delete requests. With new keys being inserted (at the front of the buckets) and existing keys being deleted, the hash table arrangement steadily becomes worse. Learning is triggered periodically which bounces back the performance. An overall gain of 5.4% is observed.



(b) Ready mostly workload (§3.6.2.4) with $zipf = 1$. We issue 98% fetch requests and 2% insert requests. Rehashing is triggered when the load factor reaches 1.5, which happens every $75 \cdot htsize$ requests. When rehashing occurs, we double the periodicity of sensing (N_S) and the duration of learning (N_L), i.e., learning is triggered less frequently for longer duration. We observe a gain of 1% in throughput.

Figure 3.12: Comparing the performance of VIP hashing to the default (vanilla) implementation of hash table when subjected to identical workloads with dynamic popularity distribution at low skew ($zipf = 1$). In Fig. 3.12a, the workload consists of evenly balanced insert and delete operations, while the workload in Fig. 3.12b consist of a small percentage of insert operations. Both the workloads are dominated by fetch operations.

3.6.2.2 Popularity Churn

In this workload, the popularity distribution shifts over time – we simulate a medium (25%) and high (50%) rate of shift every 100M (about 3s) and 10M (< 1s) requests respectively. Fig. 3.11a shows the behavior of VIP hashing under medium churn – 3 out of the 4 times when the popularity shifted, there was a substantial change in average displacement (accompanied by a decrease in performance) which was detected in the sense mode, and learning was triggered only when necessary. For the case of high churn (Fig. 3.11b), popularity shift occurs 50 times during the experiment, and every run of the sense mode detects a change in distribution and learning is triggered. We obtain a net increase of 19% and 12% in throughput for the case of medium and high churn respectively. Thus, VIP hashing is able to sense changes in distribution and re-learn on the fly.

3.6.2.3 Steady State

Next, we create a workload with 98% fetch, 1% insert, and 1% delete requests. The cardinality of the hash table doesn't change substantially during the experiment, as the number of insert and delete operations are balanced. The keys are inserted (deleted) in random positions of the popularity order. We observe that as new keys (which are less popular with high probability) are inserted at the front of the chains, the hash table arrangement steadily becomes worse and the performance of VIP hashing approaches the default. A change in average displacement is sensed every time and learning is triggered, which bounces back the performance of VIP hashing. We observe a 5.4% gain in throughput.

3.6.2.4 Read Mostly

In this workload, we issue 98% fetch requests and 2% insert requests. New keys are inserted in arbitrary positions in the popularity order. Similar to §3.6.2.3, we observe that the performance steadily becomes worse as new keys are inserted at the front of the bucket chains. Inserting new keys increase the load factor, which degrades the throughput of the default implementation as well. Rehashing is triggered when the load factor exceeds 1.5 (happens every $75 \cdot htsize$ requests), which bounces back the performance for both the default and VIP hashing implementations. The periodicity at which sensing is triggered (every $90 \cdot htsize$ requests) increases every time rehashing is performed, as we update the parameters $\mathrm{N_S}$ and $\mathrm{N_L}$ according to the size of the hash table ($htsize$). Given that the change in the distribution is substantial, every run of the sense mode detects a change in popularity and triggers learning. Overall, we obtain a gain of 1% in throughput.

3.7 Related Work

Hash tables are well studied data structures in literature. Two major categories of hash tables are chained hashing [Wika] where collisions are resolved by chaining (§3.2.1), and open addressing [Wike] where collisions are resolved by searching for alternate positions in an array. Richter et al. [RAD15] study different hash table implementations spanning both the categories, hash functions, workload patterns, etc. while highlighting the variability in the performance of hash tables based on a host of factors. Similar to our work, they consider the problem of hashing 8-byte integer keys and values.

Multiple open source hash tables [int20, Ska, BEKP18] use both categories of implementations. For instance, Google's flat hash table [BEKP18] uses open addressing, while the bytell (byte linked list) hash table [Ska] uses chaining to resolve collisions. When it comes to data systems, DBMS such as SQLite3 [sqla] and PostgreSQL [pos], as well as key-value stores such as Redis [Nat17] and Memcached [He] use data structures that involve chaining of entries. Thus, we find that chained hash tables are a popular choice commonly used in practice.

Skew in popularity is a well studied phenomenon. Multiple studies involving production workloads have found fetch requests to follow a power-law behavior [AXF+12, BCF+99], which is often captured using the zipfian distribution [BLP11, WSH19, CST+10]. For instance, the request distribution in the core workloads of YCSB [Coo10] is zipfian by default. Alongside skew in popularity, previous work [AXF+12] also discusses effects such as churn in popular keys in real world workloads. This is a key feature captured by Wiscer (§3.3), which is not present in any of the existing workload generators to the best of our knowledge.

Broadly speaking, caching algorithms such as LRU-k [OOW93] and MRU [CD05] attempt to capture the current popularity distribution. Key-value stores designed for disk-based settings, such as Anna [WSH19] and Faster [CPK+18] incorporate techniques to keep hot data in memory for better performance. Recent work by Herodotou et al. [HK19] uses machine learning (ML) to automatically move data between different storage tiers in clusters. A recurring trend to note here is that the complexity of these schemes depend on the "budget" available, ranging from simple LRU approach used even in processor caches, to a more complex approach involving ML in large-scale clusters.

The budget available for learning with hash tables is extremely limited (Fig. 3.6). In the seminal paper on learned indexes [KBC+18], the authors propose learning a hash function from the keys such that collisions can be avoided altogether. However, recent work on learned hash functions [SVH+21] shows that this approach encounters two major limitations – cache sensitivity, and model complexity. While larger models are necessary to accurately capture arbitrary key

distributions, the computation times become prohibitively high ($50x$ higher [SVH$^+$21]) due to increased cache misses from accessing the model parameters. The high cache sensitivity and low latency requirements of hash tables preclude the use of costly ML techniques.

A noteworthy aspect of the VIP hashing method is that learning is performed online, i.e., the hash table does not pause operation at any time. In contrast, recent work [SVH$^+$21, HSI22] involves learning from the data offline before populating the hash table. Adapting to changing key distributions remains a challenge with these approaches, as their fallback mechanism is reverting to the default hash table implementation [HSI22] or relearning [SVH$^+$21, KBC$^+$18], both of which require costly rehashing that pauses execution.

Chapter 4

Splitting Dataframes for Memory-Efficient Data Analysis

4.1 Introduction

Data scientists often deal with large datasets that are tabular, distributed in open formats such as CSV, JSON, and Parquet [kag]. The working environment for many data scientists are Python notebooks, either hosted on a laptop or virtualized cloud containers. Data scientists perform tasks such as data cleaning, feature engineering, and exploration using data analysis libraries such as Pandas [pan], Ibis [ibi], Koalas [koa23], etc. to name a few.

Dataframe is a key tabular construct used by data analysis libraries. While different libraries offer different abstractions for the dataframe object [PML+20], one common aspect of all dataframe libraries is that they operate on a tabular view of the data loaded directly from raw data files. Dataframes have no semantics of normalization associated with them. While loading into dataframes directly from raw data files is convenient for data analysis, dataframe libraries are known to suffer from high memory utilization [das23b, das23a, Wes17] as we also see in our work (Fig. 4.1).

A key contributor to high memory usage of data analysis libraries is redundancy in the data. Redundancy arises when the data has correlated/dependent attributes, or attributes with few unique values. Relational database systems have employed normalization to systematically identify and eliminate redundancy from the data. Designing an effective relational schema involves discovering functional dependencies in the data and taking steps to conform to a desired normal form. These steps are often performed by a database administrator. The key question we consider is: *Can specific principles of normalization from database theory be applied to dataframes to improve storage efficiency and data analysis speed, with minimal effort on the part of the data scientist?*

We describe a technique called *splitting* which is inspired by the lossless-join decomposition mechanism [RG03] from database normalization theory. Splitting can be automatically performed on tabular data, and does not require functional dependency discovery and schema design on part

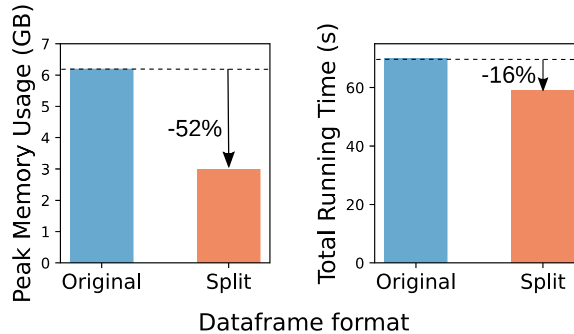


Figure 4.1: Peak memory usage of and total running time of a notebook implemented on the NYC parking tickets 2014 CSV dataset of size 1.9 GB. When operating on the original dataframe, the peak memory usage of the notebook is 3.3x the size of the raw CSV data. Operating on split dataframe reduces memory usage by 52% and improves running time by 16%.

of the user. A split dataframe internally operates on split data, while exposing the same unified tabular interface as if operating on the original data. To demonstrate the effectiveness of splitting for improving memory efficiency of dataframe libraries, we make the following contributions:

- *Formal definition of splitting*: In a nutshell, splitting involves performing a lossless-join decomposition [los] on a table by explicitly introducing joining keys (Fig. 4.2). Splitting can be performed on arbitrary groups of attributes without requiring functional dependency discovery.
- *Generating automatic splits with `SplitGen`*: We developed an algorithm `SplitGen` that generates attribute groups for splitting using statistics from the data, and does not require the user to perform schema design. We implemented `SplitGen` in Velox [PEB+22] to automatically generate split CSV files.
- *Splitting dataframes in Ibis with SplitDF*: SplitDF is an implementation of split dataframes in Ibis [ibi] for DuckDB [RM19] backend. SplitDF makes minimal changes to the Ibis API, while improving memory efficiency by operating on split data under the hood.
- *Evaluation on open datasets*: We conduct our evaluation on top-voted CSV datasets collected from Kaggle [kag]. We implemented five notebooks for the US Accidents dataset with size 1.2GB. Running the notebooks on SplitDF produces a reduction in memory usage of 19-23% and a reduction in running time of 1-25% when operating on split data as compared to operating on original data. The improvement in memory efficiency stems from the effectiveness of splitting. Running `SplitGen` on twelve open CSV datasets shows that for six out of the twelve datasets tested, we obtain more than an 40% reduction in total size from splitting.

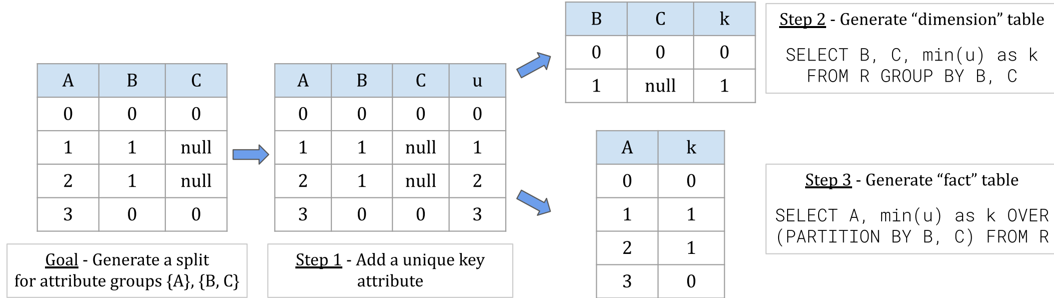


Figure 4.2: Two-way splitting of a table R = {A, B, C} into tables with schema {B, C, k} and {A, k}, where k is the joining key. In this example, the generated split satisfies the functional dependency k → {B, C}. Our proposed technique, splitting, is an efficient form of automatic lossless join decomposition, i.e., the original table can be recovered by joining the split tables on attribute k. Generating the split tables requires using aggregation and window operations. N-way splits can be generated by splitting the fact table (N-1) times.

We give a formal specification of splitting in §4.2, followed by describing `SplitDF` in §4.3. We describe the `SplitGen` algorithm in §4.4, followed by evaluation in §4.5. We discuss related work in §4.6. Through our work, we aim to demonstrate the benefits of the proposed splitting mechanism in improving the memory efficiency of dataframe libraries.

4.2 What is Splitting?

In this section, we formally define splitting in §4.2.1 followed by describing how to generate a split in a relational engine in §4.2.2. We discuss the differences between splitting and normalization in §4.2.3, and describe our vision splitting dataframes in §4.2.4.

4.2.1 Definition

Given a relation schema R, a two-way split of R into schemas with attribute sets X $\bigcup\{k\}$ and Y $\bigcup\{k\}$ is such that

- X $\bigcup$ Y = R, X $\bigcap$ Y = $\emptyset$
- $k \notin$ R
- either of the functional dependencies (FDs) $k \rightarrow$ X or $k \rightarrow$ Y hold

In other words, a two-way split is a lossless decomposition of the schema R $\bigcup\{k\}$, where the property of losslessness follows from the constraint that either $k \rightarrow$ X or $k \rightarrow$ Y hold [RG03]. The unique aspect of splitting is that we explicitly introduce a "joining" key k in the schema

that satisfies either $k \rightarrow$ X or $k \rightarrow$ Y allowing one to perform the split for any disjoint attribute groups (X, Y). Given an instance r of relation R, and a two-way split of r into x' and y' which are respectively instances of schemas X $\bigcup\{k\}$ and Y $\bigcup\{k\}$ that satisfy the above mentioned criterion, r can be recovered as $\pi_R(x' \bowtie y') = r$. A two-way split can easily be generalized to obtain an n-way split of a relation schema R.

4.2.2 Generating a Split

Fig. 4.2 shows the steps involved in generating a two-way split of a relation R = {A, B, C} into attribute groups {A} and {B, C}. The first step is to add an unique key attribute (u) to the relation which is supported by most major database engines [pos10, duc23]. To generate a split satisfying the FD k → {B, C}, the dimension table with schema {B, C, k} is generated by performing an aggregation over attributes {B, C} of R. Note that k is the primary key of the dimension table. The fact table is generated using window operation on the relation schema R $\bigcup$ {u} over attributes {B, C} . Note that both these operations involve simple aggregations. To generate an n-way split, one could recursively apply splitting to the fact table to generate *(n-1)* dimension tables. The approach shown in Fig. 4.2 can be implemented in any relational DBMS.

4.2.3 Splitting vs Normalization

Both splitting and normalization involve decomposition of a relation to reduce redundancy in the schema. However, normalization also accounts for integrity constraints of the database to guard against update, insertion and deletion anomalies. Mining functional dependencies (FDs) [PEM+15] is an important step in normalization, and these FDs are used for generating the database schema in normal forms [Cod71, Cod74]. Thus, two properties of decompositions that are of interest in the context of normalization are lossless-join and dependency-preservation [RG03].

However, guarding against insertion, update, and delete anomalies is not a primary goal of data analysis, which often involves operations such as data exploration, cleaning, and handling null values. Thus, the key property of decompositions that is of interest in the case of data analysis is lossless-join. Splitting enables exploration of attribute groups that can reduce the overall redundancy, thus improving memory-efficiency of data analysis pipelines. Unlike normalization, splitting can be performed without functional dependency discovery and schema design on part of the user.

4.2.4 Splitting Dataframes

Dataframes are popular tabular structures used in data analysis libraries. Unlike a database administrator, data scientists do not perform schema design, which would require performing functional dependency discovery and normalization. Data is directly loaded from raw tabular data files into a dataframe. Thus, we propose splitting dataframes "under the hood", i.e., exposing the same unified tabular interface to the data scientist as if loaded directly from the raw data file, while internally the dataframe operates on split data.

There are two major benefits of splitting – (1) it reduces redundancy from the data, and (2) it can be applied to raw data files using automated methods (described in §4.4) without requiring schema design. Typically, tabular data is distributed in open formats such as CSV, Parquet, and JSON [kag], and major relational database engines support loading and storing data from these formats [duc]. *Split* data files can be generated using a relational engine, where a split file is a collection of (ideally) smaller files corresponding to the fact and dimension tables generated during splitting. Thus, splitting can be performed on raw tabular data without requiring manual intervention on part of the data scientist. The data scientist can then operate on the split data by loading it into a split dataframe that exposes a unified tabular representation to the user, as if operating on the original data file.

4.3 Splitting Dataframes in Ibis

In this section, we describe salient features of the Ibis library (§4.3.1), followed by describing SplitDF, our implementation of split dataframes in Ibis for the DuckDB backend (§4.3.2).

4.3.1 The Ibis library

The Ibis [ibi] library enables working on data from over fifteen backend engines in a dataframe environment, with DuckDB being the default backend. Ibis allows working on large datasets stored in relational databases or big data systems. This is an increasingly popular trend in data science libraries, also adopted by other systems [JED+21, Hag20, SC21] to enable working on larger than memory datasets. Thus, we chose Ibis as the system of evaluation given its growing popularity as a unified dataframe interface for data analysis.

```python
import ibis

def init_from_csv(dbname, tablename, csv_file):
  """
  Load data from csv_file into table tablename
  """
  ...
  return schema

def init_from_split_csv(
    dbname, tablename, split_csv_folder):
  """
  Load the split files as individual tables,
  and declare a view with name tablename
  """
  ...
  return schema

### 1. Initializing DuckDB backend ###

dbname = "us_accidents.db"
tablename = "accidents"

### Default - Load CSV file into backend ###
# csv_file = "US_Accidents_Dec21_updated.csv"
# schema = init_from_csv(
#   dbname, tablename, csv_file)

### Split - Load split CSV into backend ###
split_csv_folder =
  "US_Accidents_Dec21_updated_split/"
schema = init_from_split_csv(
  dbname, tablename, split_csv_folder)

### 2. Register the schema with Ibis ###

con = ibis.duckdb.connect(dbname)
con.register_schema(schema)

### 3. Proceed with Data Analysis Agnostic ###
###    to the underlying format             ###

df = con.table(tablename)
count_by_state = df.group_by('State')
  .aggregate(df.count())
```

Listing 4.1: Example of data analysis with SplitDF, our implementation of split dataframes in Ibis with DuckDB backend. After initializing the backend (step 1) and registering the schema with Ibis (step 2), data analysis proceeds agnostic to the underlying storage format. The only API change SplitDF makes is registering the schema (step 2). SplitDF allows operation on both original and split CSV data with no changes to the data scientist's experience.

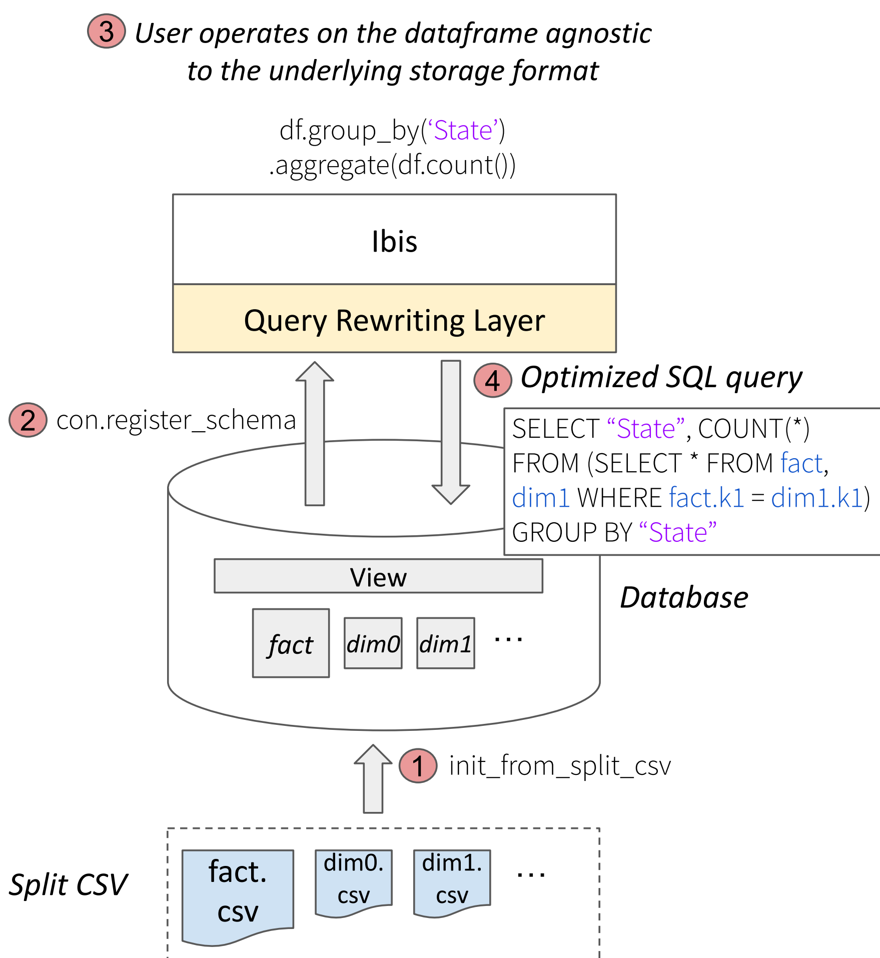


Figure 4.3: Architecture of SplitDF when operating on split data. Split CSV data is loaded into the backend database and exposed as a view to Ibis. The schema of the view is registered with the Ibis query rewriting layer (introduced in our work), which generates efficient SQL queries depending on the schema of the data (split vs unified). The user can conduct data analysis agnostic to the underlying storage format.

4.3.2 SplitDF

We develop SplitDF, an implementation of split dataframes in Ibis for DuckDB backend. The approach taken by SplitDF can be applied to Ibis for other relational backend engines as well. SplitDF has the following key features:

- *Exposing split data as a unified view*: To retain the same unified tabular dataframe interface for the user, we load the split files in the backend engine and declare a view. Ibis does not differentiate between tables and views, so the users can operate on a dataframe corresponding to the unified view agnostic to the underlying storage format.
- *The query rewriting layer*: We implemented a query rewriting layer in Ibis which transparently generates optimized SQL queries when operating on split data. The query rewriting layer maintains information about the underlying schema of the data, and generates SQL queries such that only the required dimension tables are joined with the fact table when operating on split data. While one might expect the query optimizer in the backend engine to perform said optimization, our analysis showed that the optimization is missing in prominent database engines, namely PostgreSQL and DuckDB. The query rewriting layer internally uses the SQLGlot transpiler library [sql23].
- *Minimal changes to the user experience*: The only API change made by SplitDF is registering the schema of the data with Ibis to generate efficient SQL queries using the query rewriting layer. The user can conduct data analysis on a single dataframe corresponding to the unified tabular view of the data, when when operating on split data.

Listing 4.3 shows the programmer's experience when working with SplitDF. After (1) setting up the backend, and (2) registering the schema with Ibis, (3) data analysis can proceed agnostic to the underlying layout of data. Thus, SplitDF makes minimal changes to the Ibis API. The query rewriting layer generates efficient SQL for split data by joining only the required dimension tables with the fact table as shown in Fig. 4.3. It should be noted that both the original and the split data are stored in column store format in the DuckDB backend engine of SplitDF, and optimizations made by the backend engine, such as columnar storage, apply to both original and split data.

4.4 Generating Automatic Splits

A critical aspect of splitting is the choice of attribute groups such that redundancy in the data is reduced. We propose a greedy algorithm to find attribute groups that reduces the total size of the split data (§4.4.1). We also describe an implementation of this algorithm in Velox [PEB+22] to generate split CSV files (§4.4.2).

Algorithm 6 `SplitGen`: Generating Attribute Groups for Splitting

```
 1: procedure GenAttributeGroups(t)                                    ▷ t is a table
 2:     attrs ← t.attributes
 3:     nrows ← t.nrows
 4:     distinct_count ← [CountDistinct(t[a]) for a in attrs]
 5:                                            ▷ Sorted by increasing value of distinct count
 6:     attrs ← sort(attrs, key = distinct_count)
 7:     distinct_count ← sort(distinct_count)
 8:     max_size ← [MaxValueSize(t[a]) for a in attrs]
 9:     avg_size ← [AvgValueSize(t[a] for a in attrs]
10:     stats ← (nrows, attrs, distinct_count, max_size, avg_size)
11:
12:     attr_group ← {}, dims ← [], fact ← [], i ← 0
13:     while i < len(attrs) do
14:         candidate ← attr_group.add(attrs[i])
15:         estimated_size ← EstimateSplitSize(candidate, stats)
16:         actual_size ← ActualSize(candidate, stats)
17:         if estimated_size < actual_size then
18:             attr_group ← candidate
19:             i ← i + 1                                   ▷ Try adding the next attribute
20:         else if size(attr_group) > 0 then
21:             dims.add(attr_group)
22:             attr_group ← {}                                         ▷ Start a new group
23:         else
24:             fact.add(attrs[i])                               ▷ attr[i] could not be split
25:             i ← i + 1
26:     return (dims, fact)
```

```
27: procedure ActualSize(candidate, stats)
28:     (nrows, attrs, distinct_count, max_size, avg_size) = stats
29:     pos ← attrs.get_indexes(candidate)
30:     size ← 0
31:     for i in pos do
32:         size ← size + nrows × avg_size[i]
33:     return size
34:
35: procedure EstimateSplitSize(candidate, stats)
36:     (nrows, attrs, distinct_count, max_size, avg_size) = stats
37:     pos ← attrs.get_indexes(candidate)
38:     est_nrows ← 1
39:     est_tuple_size ← 0
40:     for i in pos do
41:         est_nrows ← est_nrows × distinct_count[i]
42:         est_tuple_size ← est_tuple_size + max_size[i]
43:     est_tuple_size ← est_tuple_size + 8                    ▷ 8-byte joining key
44:     size ← est_tuple_size × est_nrows + nrows × 8
45:     return size
```

4.4.1 `SplitGen`: A greedy algorithm for split schema generation

The goal of generating splits is to reduce the redundancy in the data. We propose a greedy algorithm `SplitGen` which can be automatically executed on a table without requiring functional dependency discovery and schema design on part of the user. Instead, `SplitGen` utilizes statistics about the data to produce attribute groups for splitting. Splitting can subsequently be performed in a relational engine as shown in Fig. 4.2. Algorithm 6 describes the `SplitGen` algorithm, which generates attribute groups for the dimension tables ($dims$) and the fact table ($fact$) for splitting. The algorithm has the following components/steps:

1. *Statistics*: The `SplitGen` algorithm utilizes three key statistics about the data, namely the number of distinct values, the maximum value size, and the average value size of each of the attributes.
2. *Sliding window over attributes*: The attributes are sorted in ascending order of their distinct count. The algorithm attempts to group together attributes starting from the attribute with least number of distinct values. For each candidate attribute group, the size of the split is estimated and compared to the actual size of the attribute group. The algorithm continues to add attributes to the candidate attribute group, until the estimated split size is less than the actual size.
3. *Generating attribute groups for dimension and fact tables*: Attribute groups for which the estimated split size is less than the actual size are added to the $dims$ array, which will correspond to a dimension table. Note that dictionary encoding is a special case of this algorithm when the size of the attribute group is one. Any attributes that are not estimated to generate benefit from dictionary encoding (which is the minimal split possible) are retained in the fact table.
4. *Estimating the size of the split*: The algorithm uses a conservative estimate of the size of the generated dimension table. The cardinality of the dimension table is estimated as the product of number distinct values of each of the attributes (which is the upper limit as not all combinations of attribute values might occur in the data) times the tuple size (estimated as the sum of the maximum value size for each of the attributes, plus the size of the joining key, which is again an upper limit). Extra space needed for the joining key attribute in the fact table is also accounted for. Thus, the estimated size of the split is an upper limit on the real size of the split.

The algorithm is guaranteed to generate attribute groups for splitting that lead to a net reduction in size, since attribute groups are generated where the estimated split size is smaller than the actual size, and the size of the generated split is at most the estimated split size (we use a conservative estimate). Since the costliest step is sorting the attributes by their distinct count, the

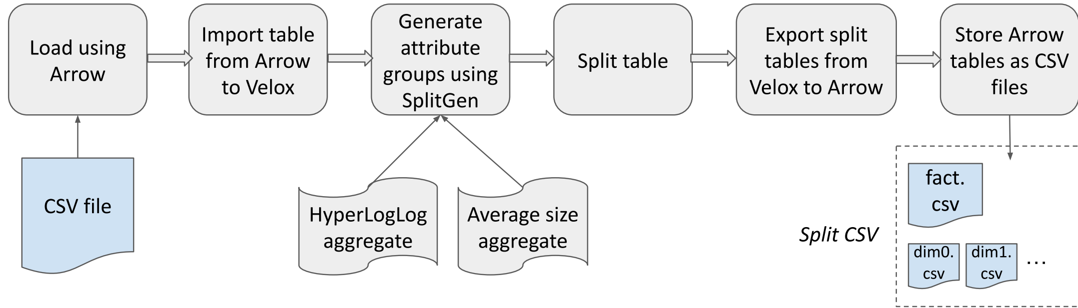


Figure 4.4: Workflow for splitting CSV files in Velox.

complexity of this algorithm is $O(a \cdot log(a))$, where a is the number of attributes in the table. While running a sliding window to generate attributes groups, each attribute is considered at most twice to be added to the candidate attribute group, and the complexity of this part of the algorithm is $O(a)$.

4.4.2 Splitting CSV files in Velox

We developed a module in Velox [PEB+22] that automatically generates split CSV files. A split CSV file is a collection of CSV files corresponding to the fact and dimension tables generated during splitting. Below are the important components of our implementation:

- *Reading/Writing CSV files using Apache Arrow*: Velox currently does not support reading/writing from CSV files. The module use Apache Arrow [arr] to ingest input CSV files and write split CSV files.
- *Implementing `SplitGen`*: The statistics utlized by `SplitGen` are obtained using aggregate functions in Velox. To estimate the distinct count of each attribute, we use the HyperLogLog [FN85, FFGM12] aggregate function in Velox. To estimate the size of the split, we relax `SplitGen` to use average value size in our implementation. Thus, the splits generated by our module are not guaranteed to be smaller, but practically we find that our module yields split CSV files that are significantly smaller in size (see §4.5.3).
- *Generating split tables*: The module uses aggregation and window operations as shown in Fig. 4.2 to generate dimension and fact tables respectively. To generate N-way splits, splitting is recursively applied to generated fact table (N-1) times.

Thus, splitting can be applied automatically to CSV files without manual intervention and schema design using the developed module in Velox.

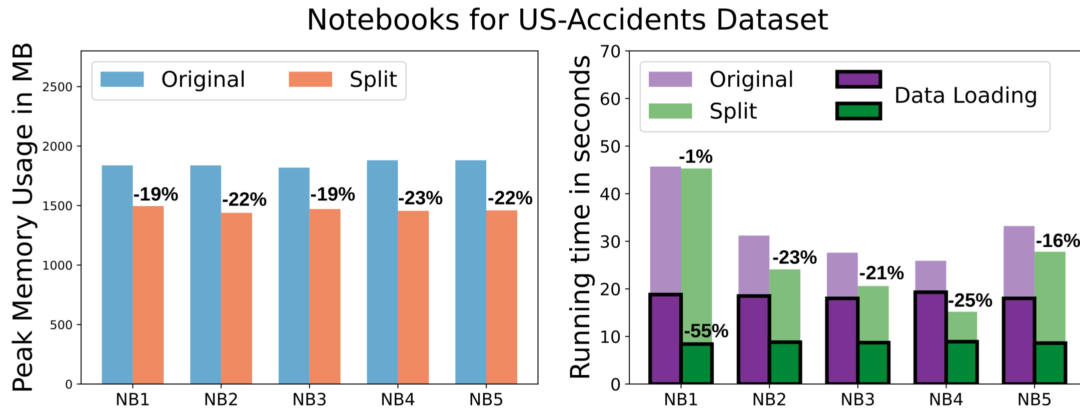


Figure 4.5: We implemented five notebooks for the US-Accidents dataset. When operating on the split dataframes, we observe a 19-23% reduction in peak memory usage across all notebooks, while the running time of the notebooks reduces by 1-25%. A large component of this reduction stems from reduction in data loading time, which is 55% lower for the split file.

4.5 Evaluation

In this section, we evaluate the performance of notebooks running on SplitDF (§4.5.2), followed by the efficiency of `SplitGen` for generating split data (§4.5.3). The evaluation is conducted on top-voted CSV datasets from Kaggle [kag] listed in Table 4.1.

4.5.1 Experimental Setup

The experiments in this section have been run on a laptop with 16GB RAM and 8 virtual cores. All experiments and microbenchmarks have been written in Python. The operating system is Ubuntu 20.04.6 LTS. Memory usage reported in all experiments is the peak resident set size during the process' lifetime, obtained using the GNU `time` [gnu] tool.

4.5.2 Running notebooks on SplitDF

Fig. 4.3 shows the architecture of SplitDF. To evaluate the effectiveness of split dataframes, we implement five Ibis notebooks for the US Accidents (1.2GB) dataset. The notebooks cover the wide range of operations typically conducted for data analysis – feature engineering, handling null values, exploratory data analysis – to name a few. All of these notebooks have been made available in our GitHub repository [git23].

The results are shown in Fig. 4.5. The split datasets for these notebooks have been generated using the implementation of `SplitGen` in Velox (§4.4.2). In all the notebooks, we find that the

Table 4.1: Top-voted CSV datasets from Kaggle. We analyze datasets with sizes ranging from 50MB to 4.8GB.

CSV Dataset Name	**Abbreviation**	**Size**
FIFA 20 complete player dataset [fif]	FIFA	51 MB
COVID-19 dataset [cov]	COVID	75 MB
Emergency - 911 Calls [e91]	911	123 MB
Brazilian E-Commerce Public Dataset by Olist [eco]	ECOMM	126 MB
Football Events	FBALL	183 MB
Data Science for Good: Kiva Crowdfunding [dsg]	DSG	233 MB
515k Hotel Reviews in Europe [hot]	HOTEL	238 MB
Bitcoin Historical Data [bit]	BITCOIN	318 MB
FitBit Fitness Tracker Data [fit]	FITBIT	338 MB
US Accidents (2016-19) [acc]	ACCIDENT	1.2 GB
NYC Parking Tickets 2014 [nyc]	NYC	1.9 GB
Flight Status Prediction (2018-19) [fli]	FLIGHT	4.8 GB

peak memory usage reduces by a significant amount of 19-23% when running on split dataframes, while the running time of the notebooks reduces by 1-25%. A large portion of the improvement in running time stems from the reduction in data loading time (55% lower), as the size of the split file for this dataset is 44% lower.

4.5.3 Splitting CSV Data with `SplitGen`

We perform splitting on twelve top-voted CSV datasets collected from Kaggle listed in Table 4.1. We have chosen a wide range of datasets with sizes ranging from 50MB to 4.8GB.

To demonstrate the efficiency of `SplitGen` algorithm, we compare the relative sizes of original vs split CSV datasets. Fig. 4.6 shows the reduction in raw data size from splitting – for six out of the twelve datasets tested, we obtain a substantial reduction in total size of more than 40% (median reduction of 39.5%), which shows the effectiveness of `SplitGen` in reducing redundancy.

To show the promise of working with split tabular data, we compare the memory footprint of three prominent libraries in the Data Science ecosystem – PyArrow [pya] (Fig. 4.7), DuckDB [RM19] (Fig. 4.9), and Pandas [pan] (Fig. 4.8) – when loading the original vs the split CSV dataset. For the three libraries, we obtain a median reduction in memory usage of 39.0%, 33.5%, and 35.2% respectively. Note that for the FLIGHT dataset, both PyArrow and Pandas run out of memory when loading the original raw data, while Pandas runs out of memory for the NYC dataset as well. Splitting reduces memory footprint, enabling Pandas and PyArrow to load these large datasets.

4.6 Related & Future Work

The issues related to high main memory usage of dataframe libraries such as Pandas [pan, Wes17] are well known, making it challenging to analyze large datasets. Consequently, there have been multiple efforts in the community to scale data analysis to larger datasets. Modin [PML+20] and Dask [das] leverage a distributed runtime such as Ray [MNW+17] to perform operations on distributed dataframes. Other efforts are Vaex [vae23] which uses lazy evaluation and memory mapping, and RAPIDS cuDF [cud23] which utilizes the GPU to enable data scientists to work with larger datasets on a single machine. The primary goal of these efforts remains scaling to larger datasets, and not necessarily optimizing memory usage [das23b, das23a] which remains an important challenge.

Inspired by the pandas API, newer dataframe APIs have been developed that run atop relational database systems to draw benefits off their performance and efficiency. The Ibis [ibi] API is supported over multiple backends ranging from in-process DBMS such as DuckDB [RM19], to cloud-based solutions such as Snowflake [DCZ+16]. Grizzly [Hag20] utilizes a transpiler to convert

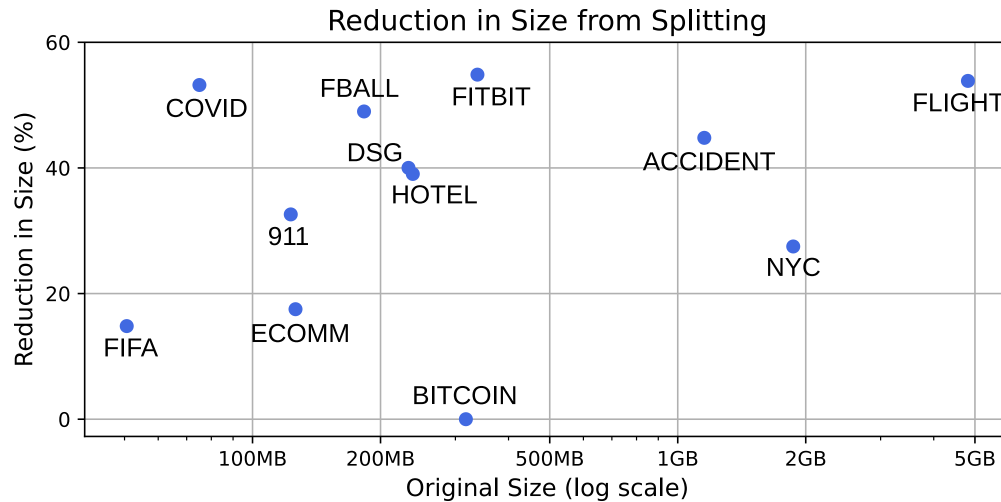


Figure 4.6: Reduction in CSV dataset size from splitting. For six out of the twelve datasets tested, we obtain a substantial reduction in size of over 40% from splitting using the `SplitGen` algorithm.

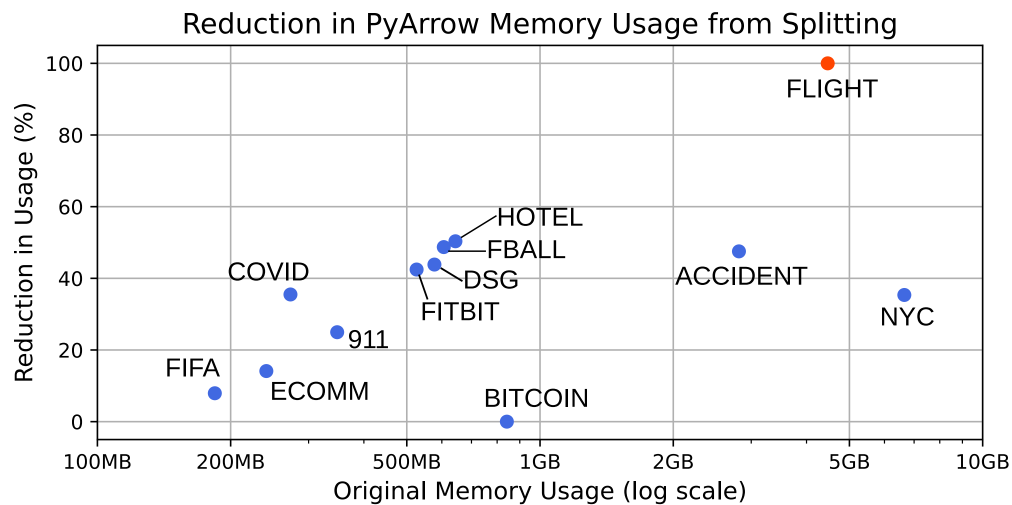


Figure 4.7: Reduction in PyArrow memory usage when loading split CSV dataset compared to loading the original dataset. For six out of twelve datasets tested, we obtain over 40% reduction in memory usage when loading split CSV datasets. For the FLIGHT dataset (marked red, in which case the x-axis indicates memory usage for the split dataset), PyArrow ran out of memory when loading the original dataset.

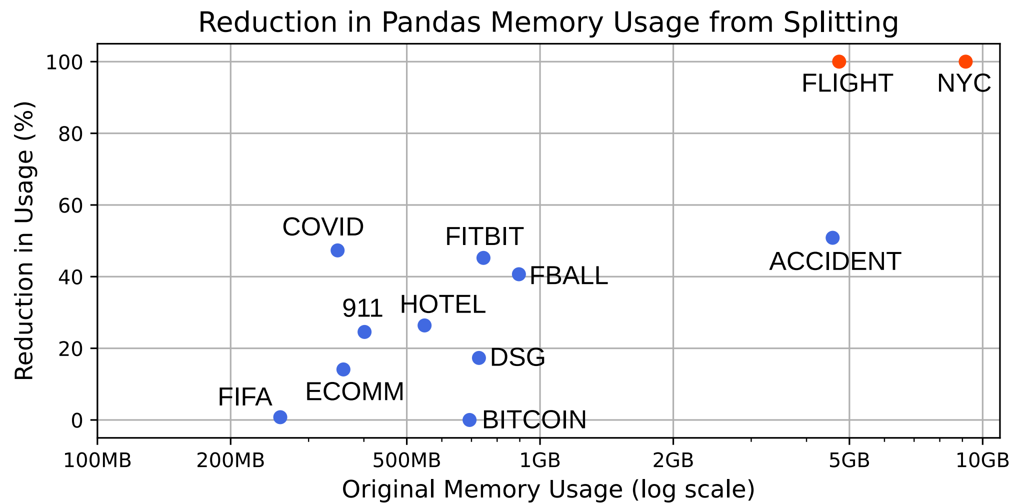


Figure 4.8: Reduction in Pandas memory usage when loading split CSV dataset compared to loading the original CSV dataset. For six out of twelve datasets tested, we obtain over 40% reduction in memory usage when loading split CSV datasets. For the FLIGHT and NYC datasets (marked red, in which case the x-axis indicates memory usage for the split dataset), Pandas ran out of memory when loading the original datasets.

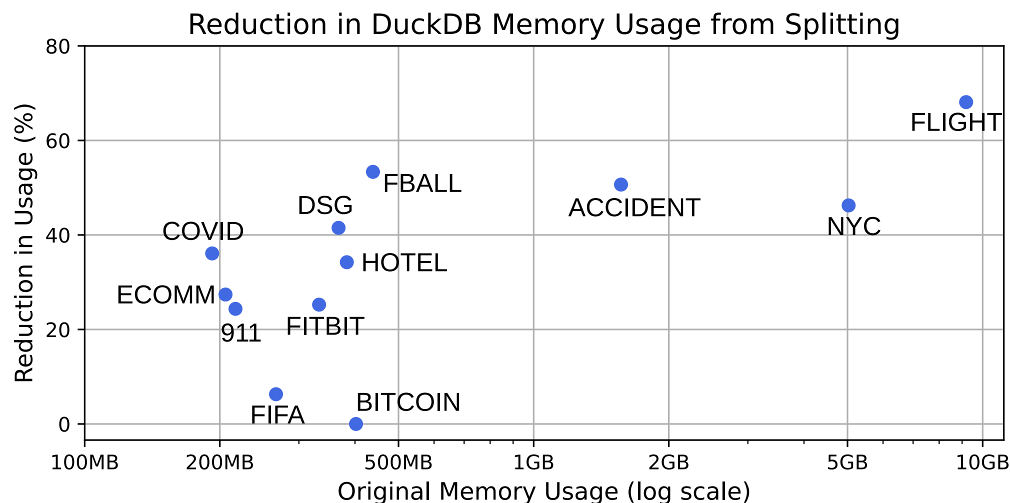


Figure 4.9: Reduction in DuckDB memory usage when loading split CSV dataset compared to loading the original CSV dataset. For five out of twelve datasets tested, we obtain over 40% reduction in memory usage when loading split CSV datasets.

pandas-like API to SQL. A similar approach is taken by the PyFroid compiler of Magpie [JED$^+$21], which converts pandas expressions into language agnostic IR utilizing Ibis, and leverages optimized relational backends in the cloud whenever possible. Other notable solutions that run atop relational DBMS are PySpark [pys23] and Koalas [koa23]. While these efforts recognize the performance and efficiency of relational DBMS, they do not leverage the fundamental optimization of lossless decomposition employed by relational DBMS.

There is a rich theory of normalization [Cod70, Cod71, Cod74, DDF12] for removing redundancy and improving the integrity of relational databases. The key distinction between normalization and splitting is that normalization requires obtaining functional dependencies from the data. Authors of [PEM$^+$15] conduct a thorough evaluation of prominent functional dependency discovery algorithms. In general, FD discovery is an expensive operation and its complexity has been shown to be $O(n^2(\frac{m}{2})^2 2^m)$ [LLLC12], i.e., quadratic in number of rows (n), and exponential in number of attributes (m). There is abundant work on finding functional dependencies, and different algorithms can broadly be classified based on finding 1) exact functional dependencies, such as DFD [ASN14], GORDIAN [SBHR06], and HyFD [PN16], and 2) approximate functional dependencies (where a dependency can be violated by only a fraction of the rows) such as PYRO [KN18], FDX [ZGR20], and TANE [HKPT99]. Although obtaining functional dependencies from the data is strictly not required for splitting, functional dependencies can help generate attribute groups with correlated/dependent attributes grouped together to further improve the efficiency of splitting. Given the expensive nature of FD discovery, techniques involving sampling and discovering correlation between columns ("soft" FDs) such as CORDS [IMH$^+$04] can potentially be used to improve split schema generation, which we aim to explore as part of future work.

Splitting can also be interpreted as a compression technique. Unlike syntactic compression techniques such as Lempel-Ziv encoding [ZL78] that treat data as a string, splitting involves extracting a structure out of the data by decomposing it into smaller fact and dimension tables. Given this motivation, splitting is similar to the work on semantic compression of relational data, where semantic relationship between the attributes is used to extract a structure for compressing data. There are two categories of semantic compression techniques – 1) lossy (i.e., allowing controlled error in attribute values), with some key techniques being ItCompress [JNOT04], SPARTAN [BGR01], and Fascicles [JMN99], and 2) lossless, such as augmented vector quantization [NR95], SInC [WSW$^+$22], and techniques proposed by Huang et al. [hua]. These semantic compression techniques can be used along with syntactic compression techniques, and are thus orthogonal. Another compression technique used in relational database systems is factorization [OS16], which involves representing query results as compact cartesian products, and is thus orthogonal to splitting as well.

Split files can be utilized as a storage format to improve storage efficiency. Another popular storage format with a similar goal is Apache Parquet [par], that divides the file into row groups, and data is stored in a columnar format within each row group. Within a row group, compression techniques such as Gzip [gzi] and Snappy [sna] are utilized to reduce space. Storing data in a columnar format improves the co-location of values of the same data type, and improves the effciency of compression. Yet, compression is performed locally in a row group, and does not utilize global patterns or correlation between attributes as one can with splitting. Storage formats such as Parquet can be utilized to further improve the storage efficiency of split files.

It should be noted that splitting dataframes in SplitDF currently benefits only the exploratory data analysis portion of notebooks. Future work can involve exploring the application of splitting to ML libraries such as CatBoost [cat] and XGBoost [CG16] to benefit the end-to-end lifecycle of a data science notebook. Splitting can also be implemented in other dataframe libraries such as pandas. Another opportunity for future work is to perform workload-aware splitting, where the knowledge of user queries can help to generate splits that reduce the number of joins performed overall, thus balancing query time along with the memory efficiency and data loading time.

Chapter 5

Conclusion

Data analytics emcompasses a wide range of tools that enable organizations to make data-driven decisions. Given the volume of data globally is increasing at an exponential rate, resource-efficiency is an important aspect of data analytics going forward. In our work, we explored three avenues of improving the resource-efficiency of fundamental data structures by utilizing the properties of the hardware, the properties of the workload, and re-evaluating the system interface.

At the lowest level, we explore how systems can efficiently utilize storage devices such as solid state drives (SSDs) by adapting to the properties of a specific device, which can lead to improvements in performance and/or increase the lifetime of the SSD. Given that SSDs are extensively used in datacenters that contain large volumes of homogeneous hardware, adapting a system to the particular make of a device can be beneficial in the present-day cloud era.

Adapting to the workload is yet another opportunity for improving the resource-efficiency of systems. In our work, we developed VIP hashing, a hash table method that utilizes lightweight learning and statistics to adapt to skew in the workload in an online manner, improving the cache-efficiency of the hash table. Over the past years, there has been ample work on developing instance-optimized database systems that utilize properties of the workload to improve the performance and resource efficiency of systems. Going forward, there is an active push from hyperscalers to integrate these learned techniques into their systems.

Finally, by re-evaluating the abstractions provided by fundamental data structures in data science, namely dataframes, we described a technique called splitting that improves the memory-efficiency of data analysis libraries. Inspired by relational database systems that extensively utilize normalization to remove redundancy from data, splitting involves automatically generating a schema by explicitly introducing joining keys. We propose splitting dataframes under the hood, i.e., continuing to provide the same unified interface as traditional dataframes, while operating on split data under the hood. Given that the initial prototype results for split dataframes in the Ibis library look promising, we look forward to the impact that this technique can have going forward.

LIST OF REFERENCES

[acc] US Accidents 2019. https://www.kaggle.com/datasets/sobhanmoosavi/us-accidents.

[AGS+09] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *PVLDB*, 2(1):361–372, 2009.

[App16] Austin Appleby. MurmurHash3. https://github.com/aappleby/smhasher/wiki/MurmurHash3, 2016.

[APW+] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, Berkeley, CA, USA.

[arr] Apache Arrow. https://arrow.apache.org/.

[ASN14] Ziawasch Abedjan, Patrick Schulze, and Felix Naumann. Dfd: Efficient functional dependency discovery. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM '14, page 949–958, New York, NY, USA, 2014. Association for Computing Machinery.

[AXF+12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. *Sigmetrics Performance Evaluation Review - SIGMETRICS*, 2012.

[Bax] Andrew Baxter. SSD vs HDD. https://www.storagereview.com/ssd_vs_hdd.

[BCF+99] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *IEEE INFOCOM '99*, 1999.

[BEKP18] Sam Benzaquen, Alkis Evlogimenos, Matt Kulukundis, and Roman Perepelitsa. Swiss Tables and absl::Hash. https://abseil.io/blog/20180927-swisstables, 2018.

[BGB17] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, 2017.

[BGR01] Shivnath Babu, Minos Garofalakis, and Rajeev Rastogi. Spartan: A model-based semantic compression system for massive data tables. SIGMOD '01, page 283–294, New York, NY, USA, 2001. Association for Computing Machinery.

[big] Google Cloud - BigQuery. `https://cloud.google.com/bigquery`.

[bit] Bitcoin Historical Data. `https://www.kaggle.com/datasets/mczielinski/bitcoin-historical-data`.

[Bla] HDD vs SSD: What does the future for storage hold? `https://www.backblaze.com/blog/ssd-vs-hdd-future-of-storage/`.

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, 2011.

[BTAO13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *2013 IEEE 29th International Conference on Data Engineering*, 2013.

[Can20] Clément L. Canonne. A short note on learning discrete distributions. *arXiv: Statistics Theory*, 2020.

[cat] CatBoost: An open-source library for gradient boosting on decision trees. `https://catboost.ai/`.

[CD05] H. Chou and D. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Algorithmica*, 2005.

[CG16] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[CGS09] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using Flash Memory to Build Fast, Power-efficient Clusters for Data-intensive Applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 217–228, New York, NY, USA, 2009. ACM.

[CKZ09] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory Based Solid State Drives. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 181–192, New York, NY, USA, 2009. ACM.

[CLZ11] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pages 266–277, Feb 2011.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, jun 1970.

[Cod71] E. F. Codd. Further normalization of the data base relational model. *Research Report / RJ / IBM / San Jose, California*, RJ909, 1971.

[Cod74] E. F. Codd. Recent investigations in relational data base systems. In *ACM Pacific*, 1974.

[Coo10] Brian F. Cooper. YCSB Core Workloads. `https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads`, 2010.

[cov] COVID 19 Dataset. `https://www.kaggle.com/datasets/imdevskp/corona-virus-report`.

[CPK+18] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *2018 ACM SIGMOD International Conference on Management of Data*, 2018.

[CPP+09] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A Survey of Flash Translation Layer. *J. Syst. Archit.*, 55(5-6):332–343, May 2009.

[cpu17] Intel Xeon Silver 4114 processor. `https://intel.ly/3fDidSb`, 2017.

[CST+10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[cud23] cudf - gpu dataframes. `https://github.com/rapidsai/cudf`, 2023.

[das] Dask. `https://www.dask.org/`.

[das23a] Dask memory limits reached in simple etl-like data transformations. https://dask.discourse.group/t/memory-limits-reached-in-simple-etl-like-data-transformations/1687, 2023.

[das23b] Tackling excessive memory usage with dask dataframes from parquet files. https://saturncloud.io/blog/tackling-excessive-memory-usage-with-dask-dataframes-from-parquet-files/, 2023.

[DCZ+16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 215–226, New York, NY, USA, 2016. Association for Computing Machinery.

[DDF12] Hugh Darwen, C. J. Date, and Ronald Fagin. A normal form for preventing redundant tuples in relational databases. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12, page 114–126, New York, NY, USA, 2012. Association for Computing Machinery.

[DJ09] Cagdas Dirik and Bruce Jacob. The Performance of PC Solid-state Disks (SSDs) As a Function of Bandwidth, Concurrency, Device Architecture, and System Organization. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 279–289, New York, NY, USA, 2009. ACM.

[DRM+19] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The Design and Operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[dsg] Data Science for Good – Kiva Crowdfunding. https://www.kaggle.com/datasets/kiva/data-science-for-good-kiva-crowdfunding.

[duc] Importing Data in DuckDB. https://duckdb.org/docs/data/overview.html.

[duc23] Create sequence in duckdb. https://duckdb.org/docs/sql/statements/create_sequence.html, 2023.

[e91] Emergency - 911 Calls. https://www.kaggle.com/datasets/mchirico/montcoalert.

[eco] Brazilian E-commerce Public Dataset by Olist. https://www.kaggle.com/datasets/olistbr/brazilian-ecommerce.

[FFGM12] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics and Theoretical Computer Science*, DMTCS Proceedings vol. AH, ..., 03 2012.

[fif] FIFA 20 complete player dataset. `https://www.kaggle.com/datasets/stefanoleone992/fifa-20-complete-player-dataset`.

[fit] Fitbit Fitness Tracker Data. `https://www.kaggle.com/datasets/arashnic/fitbit`.

[fli] Flight Status Prediction. `https://www.kaggle.com/datasets/robikscube/flight-delay-dataset-20182022`.

[FN85] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[Frø] Erik Frøseth. Hash join in MySQL 8. `https://mysqlserverteam.com/hash-join-in-mysql-8`.

[git23] Splitting. `https://github.com/UWQuickstep/splitting`, 2023.

[GKU09] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 229–240, New York, NY, USA, 2009. ACM.

[gnu] time(1) - linux manual page. `https://man7.org/linux/man-pages/man1/time.1.html`.

[gzi] GNU Gzip: General file (de)compression. `https://www.gnu.org/software/gzip/manual/gzip.html`.

[had] Hadoop. `https://hadoop.apache.org`.

[Hag20] Stefan Hagedorn. When sweet and cute isn't enough anymore: Solving scalability issues in python pandas with grizzly. In *Conference on Innovative Data Systems Research*, 2020.

[He] Holmes He. Understanding the Memcached source code. `https://holmeshe.me/understanding-memcached-source-code-V`.

[HEH+09] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman Pletka. Write Amplification Analysis in Flash-based Solid State Drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 10:1–10:9, New York, NY, USA, 2009. ACM.

[Her] Pedro Hernandez. SSD vs HDD: Price Comparison. http://www.enterprisestorageforum.com/storage-hardware/ssd-vs-hdd-price-comparison.html.

[HFVW17] F. T. Hady, A. Foong, B. Veal, and D. Williams. Platform Storage Performance With 3D XPoint Technology. *Proceedings of the IEEE*, 105(9):1822–1833, Sep. 2017.

[Hip] Richard D. Hipp. SQLite. https://www.sqlite.org/index.html.

[HJF+13] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren. Exploring and Exploiting the Multilevel Parallelism Inside SSDs for Improved Performance and Endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, June 2013.

[HK19] Herodotos Herodotou and Elena Kakoulli. Automating distributed tiered storage management in cluster computing. *Proc. of the VLDB Endowment*, 2019.

[HKADAD17] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 127–144, New York, NY, USA, 2017. ACM.

[HKPT99] Yká Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[HMvdW+20] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[hot] 515k Hotel Reviews Data in Europe. https://www.kaggle.com/datasets/jiashenliu/515k-hotel-reviews-data-in-europe.

[HSI22] Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-learned hashing: Constant time hashing with controllable uniformity. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, 2022.

[hua] Lossless Semantic Compression for Relational Databases. http://tinyurl.com/2s5ue4rp.

[ibi] The ibis project. https://ibis-project.org/.

[IDC20] IDC. Volume of data created, captured, copied, and consumed worldwide from 2010 to 2020, with forecasts from 2021 to 2025. https://www.statista.com/statistics/871513/worldwide-data-created/, 2020.

[IMH+04] Ihab F. Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, page 647–658, New York, NY, USA, 2004. Association for Computing Machinery.

[inna] The InnoDB Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[innb] The Physical Structure of an InnoDB Index. https://dev.mysql.com/doc/refman/8.0/en/innodb-physical-structure.html.

[inta] Intel Intrinsics. https://intel.ly/3nxA416.

[intb] Intel Optane SSD 900P Series. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/consumer-ssds/optane-ssd-9-series/optane-ssd-900p-series.html.

[int20] Intel TBB hash map. https://oneapi-src.github.io/oneTBB/main/tbb_userguide/concurrent_hash_map.html, 2020.

[JED+21] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Mueller, Wentao Wu, and Hiren Patel. Magpie: Python at speed and scale using cloud backends. In *Conference on Innovative Data Systems Research*, 2021.

[JMN99] H. V. Jagadish, J. Madar, and Raymond T. Ng. Semantic compression and pattern extraction with fascicles. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, page 186–198, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[JNOT04] H. V. Jagadish, Raymond T. Ng, Beng Chin Ooi, and Anthony K. H. Tung. It-compress: An iterative semantic compression algorithm. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, page 646, USA, 2004. IEEE Computer Society.

[JWZ+14] Z. Jiang, Y. Wu, Y. Zhang, C. Li, and C. Xing. AB-Tree: A Write-Optimized Adaptive Index Structure on Solid State Disk. In *2014 11th Web Information System and Application Conference*, pages 188–193, Sept 2014.

[kag] Kaggle: Your Machine Learning and Data Science Community. https://www.kaggle.com/.

[KBC+18] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, 2018.

[KJKL06] Jeong-Uk Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. A Superblock-based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 161–170, New York, NY, USA, 2006. ACM.

[KN18] Sebastian Kruse and Felix Naumann. Efficient discovery of approximate dependencies. *Proc. VLDB Endow.*, 11(7):759–772, mar 2018.

[koa23] Koalas. `https://github.com/databricks/koalas`, 2023.

[KPKP22a] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. Vip hashing – adapting to skew in popularity of data on the fly (extended version), 2022.

[KPKP22b] Aarati Kakaraparthy, Jignesh M. Patel, Brian P. Kroth, and Kwanghyun Park. VIP Hashing – Adapting to Skew in Popularity of Data on the Fly (extended version), 2022.

[KPPK20] Aarati Kakaraparthy, Jignesh M. Patel, Kwanghyun Park, and Brian P. Kroth. Optimizing databases by learning hidden parameters of solid state drives. *Proc. VLDB Endow.*, 13(4):519–532, jan 2020.

[KSJ+12] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for Solid State Disks (SSDs). *IEEE Trans. Comput.*, 61(5):636–649, May 2012.

[KTMO09] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. Application of hash to data base machine and its architecture. *New Generation Computing*, 2009.

[LHY+10] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, and Ke Yi. Tree Indexing on Solid State Drives. *PVLDB*, 3(1-2):1195–1206, 2010.

[LLLC12] Jixue Liu, Jiuyong Li, Chengfei Liu, and Yongfeng Chen. Discover dependencies from data—a review. *IEEE Transactions on Knowledge and Data Engineering*, 24(2):251–264, 2012.

[los] Lossless Join Decomposition. `https://en.wikipedia.org/wiki/Lossless_join_decomposition`.

[lsm] SQLite4 LSM Benchmark. `https://sqlite.org/src4/doc/trunk/www/lsmperf.wiki`.

[mara] MariaDB Success Stories. `https://mariadb.com/kb/en/library/mariadb-success-stories`.

[Marb] MariaDB team. Storage Engines: MariaDB Knowledge Base. https://mariadb.com/kb/en/library/storage-engines.

[mar13] MariaDB Storage Index Types. https://mariadb.com/kb/en/storage-engine-index-types/, 2013.

[MNW+17] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. *CoRR*, abs/1712.05889, 2017.

[MWKM15] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 177–190, New York, NY, USA, 2015. ACM.

[Nat17] Kousik Nath. A little internal on Redis hash table implementation. https://bit.ly/3pfVvTm, 2017.

[NK07] Suman Nath and Aman Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 410–419, New York, NY, USA, 2007. ACM.

[NR95] W.K. Ng and C.V. Ravishankar. Relational database compression using augmented vector quantization. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 540–549, 1995.

[nyc] NYC Parking Tickets. https://www.kaggle.com/datasets/new-york-city/nyc-parking-tickets/.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. In *Proceedings of the 1993 International Conference on Management of Data*, SIGMOD '93, 1993.

[opc] SQLite Testing Interface Operation Codes. https://www.sqlite.org/c3ref/c_testctrl_always.html.

[OS16] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, sep 2016.

[pan] pandas. https://pandas.pydata.org/.

[par] Apache Parquet. https://parquet.apache.org/.

[PCK+08] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications. *ACM Trans. Embed. Comput. Syst.*, 7(4):38:1–38:23, August 2008.

[PEB+22] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. Velox: Meta's unified execution engine. *Proc. VLDB Endow.*, 15(12):3372–3384, aug 2022.

[PEM+15] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery: An experimental evaluation of seven algorithms. *Proc. VLDB Endow.*, 8(10):1082–1093, jun 2015.

[PMC17] Dhathri Purohith, Jayashree Mohan, and Vijay Chidambaram. The Dangers and Complexities of SQLite Benchmarking. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 3:1–3:6, New York, NY, USA, 2017. ACM.

[PML+20] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. Towards scalable dataframe systems. *CoRR*, abs/2001.00888, 2020.

[pmu] Intel performance monitoring events. `https://perfmon-events.intel.com/`.

[PN16] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 821–833, New York, NY, USA, 2016. Association for Computing Machinery.

[pol23] Polars: Lightening-fast Dataframe Library for Rust and Python. `https://www.pola.rs/`, 2023.

[pos] PostgreSQL Database Page Layout. `https://www.postgresql.org/docs/9.3/static/storage-page-layout.html`.

[pos10] How to add an auto-incrementing primary key to an existing table in postgresql? `https://stackoverflow.com/questions/2944499/how-to-add-an-auto-incrementing-primary-key-to-an-existing-table-in-postgresql`, 2010.

[PPJ+17] Ivan Luiz Picoli, Carla Villegas Pasco, Björn Jónsson, Luc Bouganim, and Philippe Bonnet. uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, APSys '17, pages 20:1–20:7, New York, NY, USA, 2017. ACM.

[pra] PRAGMA Statements supported by SQLite. `https://www.sqlite.org/pragma.html`.

[pya] Apache Arrow Python Bindings. `https://arrow.apache.org/docs/python/index.html`.

[pys23] Pyspark documentation. https://spark.apache.org/docs/latest/api/python, 2023.

[PZK+22] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. Data science through the looking glass: Analysis of millions of github notebooks and ml.net pipelines. *SIGMOD Rec.*, 51(2):30–37, jul 2022.

[RAD15] Stefan Richter, Victor Alvarez, and Jens Dittrich. A Seven-Dimensional Analysis of Hashing Methods and Its Implications on Query Processing. *Proceedings of the VLDB Endowment*, 2015.

[Rad17] Rado Danilak. Why Energy is a Big and Rapidly Growing Problem for Datacenters. https://www.forbes.com/sites/forbestechcouncil/2017/12/15/why-energy-is-a-big-and-rapidly-growing-problem-for-data-centers/?sh=1c7ff0635a30, 2017.

[reda] Amazon Redshift. https://aws.amazon.com/redshift/.

[redb] Data types in Redis. https://redis.io/topics/data-types.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[RM19] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery.

[RPK+11] Hongchan Roh, Sanghyun Park, Sungho Kim, Mincheol Shin, and Sang-Won Lee. B+-tree Index Optimization by Exploiting Internal Parallelism of Flash-based Solid State Drives. *PVLDB*, 5(4):286–297, 2011.

[Sam] Samsung K9XXG08UXA Flash Datasheet. http://www.samsung.com/semiconductor.

[SBHR06] Yannis Sismanis, Paul G. Brown, Peter J. Haas, and Berthold Reinwald. Gordian: efficient and scalable discovery of composite keys. In *Very Large Data Bases Conference*, 2006.

[SC21] Phanwadee Sinthong and Michael J. Carey. Polyframe: A retargetable query-based approach to scaling dataframes. *Proc. VLDB Endow.*, 14(11):2296–2304, jul 2021.

[Sch97] R.R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[Ska] Malte Skarupke. Bytell hash map. https://bit.ly/3fB8NX6.

[sna] Snappy, a fast compressor/decompressor. https://github.com/google/snappy.

[sqla] SQLite hash table implementation. https://sqlite.org/src/file/src/hash.c.

[sqlb] SQLite3 Database File Format. https://www.sqlite.org/fileformat.html.

[sqlc] Working With SQLite Databases using Python and Pandas. https://www.dataquest.io/blog/python-pandas-databases/.

[sql23] The sqlglot library. https://sqlglot.com/sqlglot.html, 2023.

[SSD] Solid-state Drive. https://en.wikipedia.org/wiki/Solid-state_drive.

[str] strace. https://strace.io/.

[SVH+21] Ibrahim Sabek, Kapil Vaidya, Dominik Horn, Andreas Kipf, and Tim Kraska. When Are Learned Models Better Than Hash Functions? *CoRR*, abs/2107.01464, 2021.

[Tal] Billy Tallis. Micron 3D NAND Status Update. https://www.anandtech.com/show/10028/micron-3d-nand-status-update.

[The21] The Mathworks, Inc. *MATLAB version 9.10.0.1613233 (R2021a)*. Natick, Massachusetts, 2021.

[tpc] TPC-H Benchmark (Version 3). http://www.tpc.org/tpch/.

[UCH12] A. J. Uppal, R. C. Chiang, and H. H. Huang. Flashy prefetching for high-performance flash drives. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[vae23] Vaex.io: An ml ready fast dataframe for python. https://vaex.io/, 2023.

[wal] Write-Ahead Logging in SQLite3. https://www.sqlite.org/wal.html.

[Wes17] Wes McKinney. Apache Arrow and the “10 Things I Hate About pandas”’. https://wesmckinney.com/blog/apache-arrow-pandas-internals/, 2017.

[Wika] Wikipedia. Hash table. https://en.wikipedia.org/wiki/Hash_table.

[Wikb] Wikipedia. Hellinger’s distance. https://en.wikipedia.org/wiki/Hellinger_distance.

[Wikc] Wikipedia. Kullback-Leibler divergence. https://en.wikipedia.org/wiki/Kullback-Leibler_divergence.

[Wikd] Wikipedia. Lindeberg-Levy CLT. https://en.wikipedia.org/wiki/Central_limit_theorem#Classical_CLT.

[Wike] Wikipedia. Open addressing. https://en.wikipedia.org/wiki/Open_addressing.

[Wikf] Wikipedia. Power Law. https://en.wikipedia.org/wiki/Power_law.

[Wikg] Wikipedia. Write Amplification. https://en.wikipedia.org/wiki/Write_amplification.

[Wikh] Wikipedia. Z-test. https://en.wikipedia.org/wiki/Z-test.

[Wiki] Wikipedia. Zipf's law. https://en.wikipedia.org/wiki/Zipf's_law.

[WKC07] Chin-Hsien Wu, Tei-Wei Kuo, and Li Ping Chang. An Efficient B-tree Layer Implementation for Flash-memory Storage Systems. *ACM Trans. Embed. Comput. Syst.*, 6(3), July 2007.

[WSH19] Chenggang Wu, Vikram Sreekanti, and Joseph Hellerstein. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment*, 2019.

[WSW+22] Ruoyu Wang, Daniel Sun, Raymond Wong, Raj Ranjan, and Albert Y. Zomaya. Sinc: Semantic approach and enhancement for relational data compression. *Knowledge-Based Systems*, 258:110001, 2022.

[XSY+] Yong Xu, Kaixin Sui, Randolph Yao, Hongyu Zhang, Qingwei Lin, Yingnong Dang, Peng Li, Keceng Jiang, Wenchi Zhang, Jian-Guang Lou, Murali Chintalapati, and Dongmei Zhang. Improving service availability of cloud systems by predicting disk error. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 481–493.

[ZGR20] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. A statistical perspective on discovering functional dependencies in noisy data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 861–876, New York, NY, USA, 2020. Association for Computing Machinery.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[ZXW+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Commun. ACM*, 59(11):56–65, oct 2016.