Analytical Query Processing Beyond Main Memory

Laurens Nanno Kuiper

Analytical Query Processing Beyond Main Memory
Laurens Nanno Kuiper

Radboud Dissertation Series
ISSN: 2950-2772 (Online); 2950-2780 (Print)

Published by RADBOUD UNIVERSITY PRESS
Postbus 9100, 6500 HA Nijmegen, The Netherlands
www.radbouduniversitypress.nl

Design: Laurens Nanno Kuiper
Cover: Jonathan Auch
Printing: DPN Rikken/Pumbo

ISBN: 9789465152929
DOI: 10.54195/9789465152929
Free download at: https://doi.org/10.54195/9789465152929

RADBOUD
UNIVERSITY
PRESS

Analytical Query Processing Beyond Main Memory

Proefschrift ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.M. Sanders,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

vrijdag 21 augustus 2026
om 12.30 uur precies

door

Laurens Nanno Kuiper
geboren op 27 maart 1996
te Huntingdon, Verenigd Koninkrijk

Promotoren

Prof. dr. H.F. Mühleisen
Prof. dr. ir. A.P. de Vries

Manuscriptcommissie

Prof. dr. ir. D. Hiemstra	
Prof. dr. V. Leis	(Technische Universität München, Duitsland)
Dr. P. Tözün	(IT-Universitetet i København, Denemarken)
Dr. A. Pavlo	(Carnegie Mellon University, Verenigde Staten)
Dr. D. Porobic	(Oracle, Zwitserland)

"Life is essentially an endless series of problems.
The solution to one problem is merely the creation of another."

– Mark Manson

Acknowledgements

While studying Computer Science at university, I did not expect to find a topic I would be passionate enough about to pursue a PhD. However, I was fortunate enough to meet Arjen during my Master's, whose enthusiasm for Information Retrieval and Data Management also excited me about those topics. Collaborating with Arjen and the people I met through him shaped the course of my Master's and, eventually, my PhD. I would like to thank him and all of my ex-colleagues at Spinque, from whom I have learned so much.

Next, I would like to thank Hannes and Mark, who invited me to do a PhD in the Database Architectures group at CWI after seeing my Master's thesis defense. Although I accepted reluctantly, I have not regretted it since. Before long, they trusted me to work on the internals of their new database system, DuckDB. Through working on this system under Hannes' supervision, with Mark's extensive knowledge and code reviews, and the expertise of Peter, Stefan, and my other colleagues at CWI, I learned so much, for which I am very grateful.

During my second year at CWI, a DuckDB spin-off company was founded, which I gladly joined. This allowed me to work with even more database enthusiasts. I would like to thank all of my colleagues at DuckDB Labs for such an amazing addition to my PhD. I look forward to continuing to work with you in the future.

Finally, I would like to thank the most important people in my life. To Evi, thank you for moving across the country with me to pursue this PhD, and thank you for your love and support during each and every day that I worked on this thesis. To my parents and brother, thank you for your love and support over the years. To my friends from Tolhuis in Nijmegen, thank you for making my time at university much more fun.

Abstract

The evolution of relational database management systems has been shaped by advancements in computing hardware. Early database systems were designed around the hardware constraints of their time, optimising for limited memory, slow storage, and single-threaded execution. As hardware has advanced, offering large memory capacities, high-speed storage, and massive parallelism, database systems have adapted to utilise these resources. However, when queries require more space for temporary data than fits in main memory, database systems often suffer sharp performance drops. Due to large differences between modern *in-memory* query processing and traditional *larger-than-memory* (also referred to as *external*) query processing, execution times can increase by orders of magnitude when the memory limit is exceeded. This thesis aims to close this gap by designing query execution such that *performance degrades gracefully*, rather than *abruptly*, as the amount of temporary data exceeds the memory limit, but *without sacrificing in-memory performance*.

Database systems implement many query operators that use various algorithms and data structures. Due to the differences between operators, tackling external query execution as a whole is infeasible. Therefore, we approach this problem by addressing one operator at a time. Despite their differences, the operators we focus on have in common that they are *blocking*, i.e., they must *temporarily store intermediate relational data*. Therefore, we investigate how to store and manage these intermediates for efficient in-memory and external query processing, starting with the *sort* and *grouped aggregation* operators. Finally, we study the *join* operator and how to manage the combined memory usage of multiple operators within a query plan.

For the *sort* operator, we research how *data orientation* affects query performance. We build on prior work that showed that a row-oriented (as opposed to a column-oriented) data layout yields superior performance for grouped aggregation and joins. We demonstrate that the same holds for the sort operator. Then, we show that, with a minor modification that has a negligible impact on in-memory performance, the data stored in this row-oriented layout can be written to (and read from) storage *without fully (de-)serialising* it. Avoiding serialisation overhead reduces the performance difference between in-memory and external query processing.

For the *grouped aggregation* operator, we research how to manage intermediate data. We identify shortcomings in traditional memory management and propose to use *paged memory* to store intermediates, and unify the memory management of temporary and persistent data. Then, we propose a *buffer page layout* for intermediates that builds on the previously proposed row-oriented data layout, improving it so that it can be written to storage *without any additional serialisation costs*. We integrate the proposed techniques into the grouped aggregation operator and show that this enables larger-than-memory query processing with little additional operator complexity.

For the *join* operator, we research how to manage the memory of query plans. Unlike the sort and grouped aggregation operators, the join operator has two inputs, only one of which is *blocking*; therefore, multiple operators can be active simultaneously. If multiple operators are active simultaneously, the memory of the *entire query plan* must be managed, as the combined memory usage of the active operators cannot exceed the limit. We first integrate the previously proposed techniques for managing intermediate data into the join operator. Then, we propose a technique to dynamically assign memory to active join operators that maximises data throughput. We show that the proposed techniques allow for efficient external join processing, even in query plans with many joins.

This thesis contributes to external query processing by allowing database systems to robustly process larger-than-memory intermediates without sacrificing in-memory performance. This increases the volume of data that can be efficiently processed on typical hardware.

Samenvatting

De ontwikkeling van relationele databankbeheersystemen is sterk gevormd door ontwikkelingen in computerhardware. Vroege databanksystemen werden ontworpen rond de beperkingen van de hardware van hun tijd: weinig geheugen, trage opslag en "single-threaded" uitvoering. Naarmate hardware verbeterde, met meer geheugen, snellere opslag en meer parallelle verwerking, pasten databanksystemen zich aan om deze middelen te benutten. Wanneer query's echter meer tijdelijke data nodig hebben dan in het werkgeheugen past, kunnen scherpe prestatiedalingen optreden. Door de grote verschillen tussen moderne *in-geheugen* en traditionele *groter-dan-geheugen* (ook wel *externe*) queryverwerking, kunnen uitvoeringstijden met meerdere ordes van grootte toenemen zodra de geheugenlimiet wordt overschreden. Dit proefschrift richt zich op het verkleinen van deze kloof door queryuitvoering zo te ontwerpen dat *prestaties geleidelijk afnemen*, in plaats van *plotseling*, wanneer de hoeveelheid tijdelijke data de geheugenlimiet overschrijdt, maar *zonder in-geheugen prestaties op te offeren*.

Databanksystemen implementeren veel query-operatoren met diverse algoritmen en datastructuren. Door de verschillen tussen operatoren is het onhaalbaar om externe queryverwerking als geheel aan te pakken. Daarom benaderen we dit probleem per operator. Ondanks hun verschillen hebben de operatoren waarop wij ons richten gemeen dat ze *blokkerend* zijn: ze moeten *tijdelijk data opslaan*. Daarom onderzoeken we hoe deze tijdelijke data efficiënt kunnen worden opgeslagen en beheerd voor zowel in-geheugen als externe queryverwerking, beginnend met de *sorteer-* en *gegroepeerde aggregatie*-operatoren. Ten slotte bestuderen we de *join*-operator en hoe het gecombineerde geheugengebruik van meerdere operatoren binnen een queryplan kan worden beheerd.

Voor de *sorteer*-operator onderzoeken we hoe *dataoriëntatie* de queryprestaties beïnvloedt. We bouwen voort op eerder werk dat liet zien dat een rijgeoriënteerde indeling betere prestaties oplevert voor gegroepeerde aggregatie en joins dan een kolomgeoriënteerde indeling. Wij tonen aan dat hetzelfde geldt voor sorteren. Vervolgens laten we zien dat deze rijgeoriënteerde data, met een kleine aanpassing die verwaarloosbare invloed heeft op de in-geheugen queryverwerking, naar opslag kan worden geschreven en daaruit kan worden gelezen *zonder volledige (de-)serialisatie*. Het vermijden van serialisatiekosten verkleint het prestatieverschil tussen in-geheugen en externe queryverwerking.

Voor de *gegroepeerde aggregatie*-operator onderzoeken we hoe tijdelijke data beheerd kunnen worden. We identificeren tekortkomingen in traditioneel geheugenbeheer en stellen voor om *gepagineerd geheugen* te gebruiken voor tijdelijke data, en zo het geheugenbeheer van tijdelijke en persistente data te verenigen. Daarna stellen we een *bufferpagina-indeling* voor tijdelijke data voor, die voortbouwt op de eerder voorgestelde rijgeoriënteerde indeling en deze zo verbetert dat data naar opslag kan worden geschreven *zonder bijkomende serialisatiekosten*. We integreren de voorgestelde technieken in de gegroepeerde aggregatie-operator en laten zien dat dit groter-dan-geheugen queryverwerking mogelijk maakt met weinig extra complexiteit in de operator.

Voor de *join*-operator onderzoeken we hoe het geheugen van queryplannen kan worden beheerd. In tegenstelling tot de sorteer- en gegroepeerde aggregatie-operatoren heeft de join-operator twee invoeren, waarvan er slechts één *blokkerend* is; daardoor kunnen meerdere operatoren tegelijk actief zijn. Wanneer meerdere operatoren tegelijk actief zijn, moet het geheugen van het *gehele queryplan* worden beheerd, omdat het gecombineerde geheugengebruik van de actieve operatoren de limiet niet mag overschrijden. We integreren eerst de eerder voorgestelde technieken voor het beheer van tijdelijke data in de join-operator. Daarna stellen we een techniek voor om dynamisch geheugen toe te wijzen aan actieve join-operatoren, zodat de datadoorvoer wordt gemaximaliseerd. We laten zien dat de voorgestelde technieken efficiënte externe join-verwerking mogelijk maken, zelfs in queryplannen met veel joins.

Dit proefschrift draagt bij aan externe queryverwerking door databanksystemen in staat te stellen tijdelijke data die groter zijn dan het werkgeheugen robuust te verwerken zonder de prestaties van in-geheugen queryverwerking op te offeren. Hierdoor kan op gangbare hardware een grotere hoeveelheid data efficiënt worden verwerkt.

Contents

1. Introduction

In 1970, Edgar F. Codd proposed the relational model for data management [1], creating the foundation for databases as we know them today. In the same year, IBM released the System/370, its first computer to use semiconductor memory for its main memory. It contained one or more 48 KB memory units, each approximately 34×23×14 cm[1]. Today, a RAM stick with 512 GB of memory is available, with the regular size of approximately 14×5×1 cm[2]. This represents a bytes/cm^3 increase of more than 1,500,000,000×. The System/370 was intended to be used with one or more of the 3330 Series Disk Storage, a hard disk drive (HDD) with a capacity of 100 MB and a size of approximately 38×38×15 cm[3]. Today, a solid-state drive (SSD) with 100 TB of storage is available, approximately 15×10×3 cm[4]. This represents a bytes/cm^3 increase of almost 50,000,000×.

[1] E. F. Codd (1970), "A relational model of data for large shared data banks"

[1] Around the size of a shoebox.

[2] Around the size of a smartphone.

[3] Around the size of a large Dutch oven.

[4] Around the size of an A6 hardcover notebook.

Since their inception in the 1970s, relational database management systems (RDBMS) have been developed in an ever-changing environment. When performance is critical, developers tune software to the available hardware. An early example is the *Elevator algorithm*, which schedules disk access to align with the physical motion of the hard disk's arm [2]. Unlike implementations for *specific* use cases, RDBMS are *generic* solutions used in many different environments and on all kinds of hardware. As hardware improves, software naturally benefits. However, database systems must adapt. After all, the first system to implement the Structured Query Language (SQL) [3], System R [4], cannot achieve the performance of current systems by using modern hardware, as it would not be able to fully utilise the hardware's capabilities. Changes in hardware and database users' needs, such as the desire to process different workloads or larger volumes of data, create an ongoing need for innovation.

[2] Peter J. Denning (1967), "Effects of scheduling on file memory operations"

[3] Donald D. Chamberlin et al. (1974), "SEQUEL: A structured English query language"

[4] M. M. Astrahan et al. (1976), "System R: Relational Approach to Database Management"

By relying on principles like *memory hierarchy* (depicted in Figure 1.1) and *Amdahl's law*[5] [5] to guide the design of their systems, database developers can, to some extent, future-proof their systems. Some systems, like PostgreSQL [6], allow tuning settings to the underlying hardware by manually configuring settings[6]. However, relative changes in, *e.g.*, single-threaded and many-core performance, memory and storage latency/throughput, or hardware prices, can necessitate large architectural changes in database systems for them to make the most out of the hardware. An example of such a change in the past is increased availability of main memory in the 1990s, which caused database systems to opt for *hash-based* algorithms (that have a time complexity of $O(n)$ but must fit in main memory) rather than *sort-based* algorithms (that have a time complexity of $O(n \log n)$ but do not have strict memory requirements) [7]. Another example is the introduction of many-core CPUs in the 2000s, which caused systems to shift their parallelism frameworks from *plan-based* [8] to *data-driven* [9], which balances workloads more evenly over threads. Current developments include the rising popularity of ARM CPU architecture, which uses a different instruction set than the dominant x86 CPU architecture [10], and a relatively large increase in the input/output (I/O) speeds and parallelism of modern storage devices.

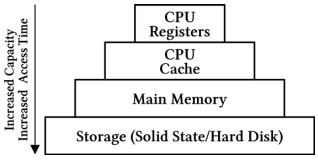

Figure 1.1 Memory hierarchy. Capacity is smaller and access time is faster at the top, while capacity is larger and access time is slower at the bottom.

[5] "The overall performance improvement gained by optimising a single part of a system is limited by the fraction of time that the improved part is actually used."

[5] Gene M. Amdahl (1967), "Validity of the single processor approach to achieving large scale computing capabilities"

[6] Michael Stonebraker et al. (1986), "The design of POSTGRES"

[6] Notably, the `seq_page_cost`/`random_page_cost` settings to tune the cost model to the storage device.

[7] David J DeWitt et al. (1984), "Implementation Techniques for Main Memory Database Systems"

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[10] Emily Blem et al. (2015), "ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures"

Because the early database systems of the 1970s operated in low-memory environments, they had to rely on external storage not only for persistent data, but also for temporary intermediates while processing queries. SSDs did not yet exist, and the lookup speed and throughput of HDD storage devices were almost always the performance bottlenecks. Hard disks largely serialise data access due to mechanical seek and rotation, limiting the effectiveness of parallel query processing. As a result, these systems optimised their algorithms for disk access, and we now consider these to be *disk-based* database systems. Once larger amounts of main memory became more readily available in the 1990s, database systems could store more persistent data and temporary intermediates in memory. Together with the speed of modern CPUs, the bottleneck of query processing has shifted to memory access, provided that data fits in memory and systems optimise their algorithms for main memory query processing [11]. The benefits of parallelism have become more relevant as main memory, unlike HDDs, can be accessed in parallel. These developments in hardware, alongside a vast body of research [7, 12], led to the emergence of *main-memory* database systems as we know them today.

[11] Peter A. Boncz et al. (1999), "Database Architecture Optimized for the New Bottleneck: Memory Access"

[7] David J DeWitt et al. (1984), "Implementation Techniques for Main Memory Database Systems"

[12] H. Garcia-Molina et al. (1992), "Main Memory Database Systems: An Overview"

A dichotomy between in-memory and larger-than-memory (external) query processing had emerged. As long as data fits in main memory, query processing is fast due to high throughput, high degrees of parallelism, and hash-based algorithms. If data does not fit, *however*, query processing can be orders of magnitude slower due to the use of sort-based algorithms and the single-threaded I/O of HDDs. These large performance differences are undesirable, as they cause query execution time to be unreliable. Luckily, a breakthrough in storage was on the horizon. Flash memory, which was invented in 1980 and is a key component of modern SSDs, was slowly making its way into the market. SSDs became widespread during the 2010s, and were quickly recognised as a viable alternative to HDDs for database systems [13]. Today, compared with HDDs, SSDs offer more than 30× the throughput and over 50× lower access latency, while also allowing a high degree of parallel data access.

[13] Sang-Won Lee et al. (2008), "A Case for Flash Memory SSD in Enterprise Database Applications"

In the 2010s, Intel Optane *persistent memory* was considered a promising technology for database systems. Optane allows for fine-grained data access, as it is *byte-addressable* like main memory, unlike SSDs, which are *block-addressable*. Optane provides better performance than SSD for database systems when used as a drop-in replacement [14], but requires the use of specialised primitives [15] to ensure ACID [7] guarantees. In combination with its considerably higher cost than SSDs, this constrained Optane's adoption. In 2021, Optane was discontinued due to a lack of profitability. Therefore, optimising database systems for the capabilities of modern SSDs is the way forward to advance external query processing.

[14] Maximilian Böther et al. (2021), "Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs"

[15] Alexander van Renen et al. (2019), "Persistent Memory I/O Primitives"

[7] ACID, which stands for *atomicity*, *consistency*, *isolation*, and *durability*, is a set of properties of database transactions intended to guarantee data validity despite errors, power failures, and other failures.

With modern SSDs, the performance difference between main memory and storage is smaller than it used to be, but many software applications have not yet adapted to this development. We have arrived at a point in time where database developers must again rethink

the algorithms and data structures used in query processing to fully utilise the capabilities of modern hardware. In this thesis, we study the most common *blocking* relational query operators, *sorting*, *grouped aggregation*, and *joins*. Blocking operators operate on entire inputs, as opposed to *streaming* operators, such as *scan*, *filter*, and *project*, which operate on individual tuples. Therefore, blocking operators are *memory-intensive*. This thesis explores how the implementations of these operators can be adapted for external query processing, to reconcile the dichotomy between in-memory and larger-than-memory processing. Finding a way to do this *without sacrificing in-memory performance*, and while *gracefully degrading performance* (as opposed to a sudden and large drop in performance) as the data size exceeds the memory limit, are the main challenges.

1.1 Outline and Contributions

The work presented in this thesis builds on decades of query-processing research; therefore, understanding key concepts is crucial. In Chapter 2, we explain the fundamentals of query processing. We discuss query execution models, parallelism, and algorithms for in-memory and external query processing.

In the chapters that follow Chapter 2, we present the research carried out for this thesis. We tackle the problem of external query processing, specifically for analytical (OLAP) database systems[8], by studying one relational query operator per chapter. Every operator uses different algorithms and data structures, which create unique challenges when implementing larger-than-memory processing. However, these operators also share important similarities, and lessons from one operator transfer to the next. The research presented in this work has been implemented in DuckDB [16], an open-source, column-oriented relational database management system.

[8] *OLAP* systems are built to analyse data, as opposed to *OLTP* systems, which are built for transaction-oriented applications.

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

Data Orientation. To evaluate a query, database engines generate a query plan. A query plan typically consists of a modest number of operators, which can process an *unbounded* number of tuples. Although query planning can cause overhead when processing queries [17], this overhead is often overshadowed by the overhead that occurs when processing tuples during query plan evaluation, as each overhead adds up when processing, *e.g.*, billions of tuples.

[17] Viktor Leis et al. (2015), "How Good Are Query Optimizers, Really?"

Two notable overheads in traditional RDBMS [9] that have been studied in depth are interpretation overhead [18] and cache misses [11]. *Interpretation overhead* comes from database systems having to implement generic code to evaluate any query plan, with any number of columns, and with any data type. As a result, a tuple's *shape*, *i.e.*, its columns and their types, is not known at compile time, requiring tuples to be *interpreted dynamically* at runtime and preventing compiler optimisations such as *loop pipelining*, which leads to low CPU utilisation. *Cache misses* are instances where the CPU cannot find requested data in *cache*

[9] By *traditional RDBMS*, we refer to disk-based relational database systems designed around tuple-at-a-time execution. This will be explained in more detail in Chapter 2.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[11] Peter A. Boncz et al. (1999), "Database Architecture Optimized for the New Bottleneck: Memory Access"

(top of the memory hierarchy, with a low access time), and has to fetch it from elsewhere (lower in the memory hierarchy, with a higher access time), causing the CPU to stall while waiting for the data[10]. In Chapter 3, we research how *data orientation*, which refers to how tabular data is represented in a linear memory model (storage or main memory), *e.g.*, *row-* or *column-oriented*, affects both of these overheads when sorting relational data. Prior work has already researched data orientation for hash joins and hash aggregation [19], and found that a row-oriented tuple layout in hash tables yields superior performance.

[10] See Figure 1.1 for an illustration of the memory hierarchy.

[19] Marcin Zukowski et al. (2008), "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing"

Although many database systems prefer to use hash-based algorithms when possible, sorting remains crucial to many database operations[11]. Sorting is a well-studied topic in computer science, and it has yielded efficient sorting algorithms. However, most of the research on sorting is limited to sorting arrays of integers in main memory; therefore, it does not take into account the per-tuple overheads seen in query processing. Furthermore, sorting relational data has many challenges that are not present when sorting an array of integers in main memory. Relational data can potentially (1) have multiple columns to sort by, (2) have many other columns that must be reordered alongside it, and (3) be larger than memory. In Chapter 3, we study how per-tuple overheads affect sorting relational data. We find that a specific row-oriented tuple layout reduces both cache misses *and* interpretation overhead. Then, we show how a row-oriented tuple layout can be used for external sorting.

[11] In SQL, sorting is explicitly invoked by the `ORDER BY` and `WINDOW` clauses, but also often occurs implicitly, for example through `CREATE INDEX` or `JOIN` (sort-merge) clauses.

Chapter 3 is based on the following publications:

- Laurens Kuiper, Mark Raasveldt and Hannes Mühleisen. "Efficient External Sorting in DuckDB". in: *Proceedings of BICOD 2021*. 2021
- Laurens Kuiper and Hannes Mühleisen. "These Rows Are Made for Sorting and That's Just What We'll Do". In: *Proceedings of ICDE 39*. 2023

The main contributions of Chapter 3 are the following:

- We present a comprehensive experimental evaluation of how data orientation affects relational sorting performance. We show that a row-oriented layout is more efficient, regardless of whether systems use vectorisation or code generation.
- We measure how interpretation overhead affects comparisons, which is one of the main costs of sorting. We show that compiled query engines are efficient, while interpreted engines suffer from various overheads. We then show that these overheads can be overcome with existing techniques.
- We propose to use *pointer swizzling*[12], rather than full (de-)serialisation, to spill in-memory data with pointers to storage. We then read the data back into memory, potentially to a different address, without invalidating the pointers. This allows what is essentially an in-memory merge sort implementation to sort data that is larger than memory.

[12] *Pointer swizzling* is the in-place conversion of references based on name or position into direct pointer references.

Spilling Intermediates. As explained earlier in this chapter, there is a dichotomy between in-memory and external query processing. Many database systems are unable to continue using hash-based algorithms for joins and aggregations when intermediate query data no longer fits in memory. Instead, they have to sort the data, causing performance to degrade sharply. Even if systems implement hash-based algorithms that can process larger-than-memory data, such as Hybrid Hash Join [22], there is still a large gap between the efficiency of in-memory and external query processing. We identify three shortcomings in traditional database management systems that cause this discrepancy: ① inefficient memory management for query intermediates, ② coarse-grained spilling of large amounts of query intermediates, and ③ (de)-serialisation overhead when converting data from its in-memory format to its storage format. While our approach to sorting larger-than-memory data in Chapter 3 addresses shortcoming ③ by avoiding a full (de-)serialisation when spilling to and reading from storage, it does not address the other two shortcomings.

[22] Leonard D. Shapiro (1986), "Join Processing in Database Systems with Large Main Memories"

In Chapter 4, we research how the three shortcomings mentioned above can be overcome, this time through the lens of the aggregation operator. Although most aggregation queries reduce the size of the input such that the result can fit in main memory, some *grouped* aggregations require spilling. Predicting whether intermediates will fit before executing a query cannot be done reliably; therefore, implementations must adapt to the workload during execution. To address these shortcomings, we propose a novel approach to memory management that is efficient for analytical workloads, which stores *temporary* query data on pages[13]. Again, we store tuples using a row-oriented layout, but we improve the pointer swizzling approach proposed in Chapter 3, allowing intermediate query data to be spilled without pointer swizzling. We show the efficacy of this approach by integrating it into an external hash aggregation algorithm.

[13] *Pages* are fixed-size buffers that are moved between memory and storage, generally only used for *persistent* data.

Chapter 4 is based on the following publication:

- Laurens Kuiper, Peter Boncz and Hannes Mühleisen. "Robust External Hash Aggregation in the Solid State Age". In: *Proceedings of ICDE 40*. 2024

The main contributions of Chapter 4 are the following:

- We present a *unified* approach to memory management that permits variable-size pages, and stores pages for temporary query intermediates in the same pool as persistent data, allowing the buffer manager to evict both to storage as needed.
- We present a novel *page layout* for temporary query intermediates on buffer pages, optimised for in-memory performance, and spillable to storage without additional serialisation overhead.
- We integrate the above two techniques into a parallel hash aggregation that can *gracefully degrade performance* as the memory limit is exceeded.

Query Plan Memory Management. When a query operator processes more data than fits in memory, the database system must manage the memory usage of the operator, as it cannot exceed the memory limit. This entails that the database system must assign memory for the operator to use. In turn, the operator must adhere to the given assignment, spilling data to storage as needed. The two operators that we have covered in Chapter 3 and Chapter 4, sorting and aggregation, are *unary*, *i.e.*, they have a single input relation, like most relational query operators. Managing the memory of these operators is simpler than for *binary* operators, *i.e.*, operators with two input relations, as only one query operator is actively using memory when evaluating a unary operator[14]. The database system can assign all the memory available for the query to the single active operator.

[14] Technically, both the query operator producing tuples, *e.g.*, a scan, and the query operator receiving them, *e.g.*, an aggregation, use memory. For simplicity, we assume that the producing operators use negligible memory.

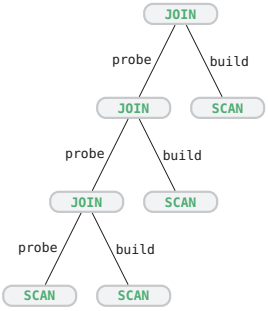

Figure 1.2 Query plan where three hash joins are probed simultaneously in a single pipeline.

The join, however, is a *binary* operator. For hash joins, the inner (build) relation is fully materialised, while the outer (probe) relation can be streamed through the operator. This implies that multiple query operators can be active simultaneously. We show an example of this in Figure 1.2. In the example, three hash joins are probed in succession. Now, the *combined memory usage* of all active operators cannot exceed the memory limit. For queries whose intermediates exceed the memory limit, the *distribution* of memory over operators affects how efficiently the query plan is evaluated. Some distributions can cause much more data to be spilled than others.

In Chapter 5, we research external join processing. First, we integrate the memory management techniques proposed in Chapter 4 into DuckDB's hash join, allowing it to process larger-than-memory query intermediates. Then, we study how and when hash joins contend for shared resources and propose a method to efficiently and dynamically distribute the available memory over active operators.

Chapter 5 is based on the following publication:

- Laurens Kuiper, Paul Groß, Peter Boncz and Hannes Mühleisen. "Saving Private Hash Join". In. 2025

The main contributions of Chapter 5 are the following:

- A *parallel* and external hash join that integrates *unified memory management* and the *spillable page layout* for intermediates, which *gracefully degrades performance* as the memory limit is exceeded.
- A dynamic *multi-operator memory control* approach that efficiently distributes the available memory over simultaneously active operators, with a *low synchronisation overhead* to prevent the performance degradation of parallel query execution.
- An optimiser that uses statistics to create expressions to *compress* and *decompress* columns *during execution*, reducing the space requirement of operators without the need to modify them.

Conclusion. After presenting our research, we conclude with our final remarks and a discussion of future work in Chapter 6.

2. Background

When a user issues a SQL query, it is first parsed, creating a *parse tree*. The parse tree is then bound, and a *logical plan* is created that specifies which relational operators to use to evaluate the query. Optimisations are then applied to create a more efficient query plan. The logical plan is finally converted to a *physical plan* (containing the algorithms to evaluate each relational operator), which can be *executed*. This chapter explains some of the fundamental concepts of *physical query execution* that the remainder of this thesis builds upon.

2.1 Streaming Query Execution

The *tuple-at-a-time iterator model*, originally implemented in the Volcano database system [25], describes a *streaming* query evaluation model. Data is recursively *pulled* through the operators of a query plan, starting from the root operator, one *tuple at a time*. Some operators, *e.g.*, filters and projections, process tuples individually, allowing data to stream through. Other operators, such as sorting and aggregation, require the entire input and are, therefore, *blocking*, which means that they cannot output tuples until all input tuples have been read. Although data cannot be streamed through blocking operators, they still fit in the tuple-at-a-time iterator model: pulling one tuple from a blocking operator will cause it to pull all tuples from the operator below. It will then finish processing, *e.g.*, by sorting the input and outputting the requested tuples, one by one. The hash join is a special blocking operator, as it has two inputs: the *inner (build) relation* and the *outer (probe) relation*. The former tends to be much smaller than the latter. Hash joins only block the inner relation, while the outer relation can be streamed.

[25] G. Graefe (1994), "Volcano-An Extensible and Parallel Query Evaluation System"

We will now distinguish between *operator-at-a-time* and *pipelined* execution, where the latter implements a streaming model. Consider, for example, the query in Listing 2.1 with the corresponding query plan shown in Figure 2.1. With streaming query execution, tuples are pulled through the entire `SCAN–FILTER–AGGREGATE` *pipeline* one by one, until all of `lineitem` has been scanned. Therefore, streaming execution is also called *pipelined* execution. Tuples travel from source operators, such as table scans, through streaming operators, to *pipeline breakers*, *i.e.*, blocking operators. Using this model, the space required to evaluate the example query plan is roughly the size of one tuple. Streaming query execution reduces the space[1] required to evaluate queries. Evaluating the same query plan using *operator-at-a-time* query execution would have a much higher space requirement, as the `SCAN` operator would be fully evaluated first, which would require loading, and, therefore, fitting all of the `lineitem` table into memory.

Listing 2.1 Example query where streaming query execution has a lower space requirement than operator-at-a-time query execution.

```
SELECT sum(l_extendedprice)
FROM lineitem
WHERE l_suppkey = 42;
```

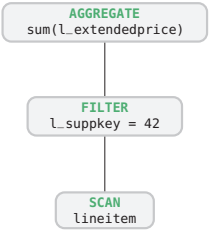

Figure 2.1 Query plan for the example query in Listing 2.1.

[1] With space we mean memory and external storage.

Another benefit of pipelined query execution is its favourable data access pattern. Multiple operators are evaluated subsequently on the same tuple, allowing the tuple to stay at the top of the memory hierarchy[2], *i.e.*, CPU registers and caches, throughout the pipeline. When evaluating one operator at a time, all of the data would move from

[2] See Figure 1.1 for an illustration of the memory hierarchy.

main memory to the CPU cache and back again for every operator. A downside of the tuple-at-a-time iterator model, however, is that while it has better cache behaviour, the recursive function call stack it uses to pull one tuple at a time through operators causes frequent control-flow changes and inhibits instruction-level parallelism, leading to instruction-pipeline stalls and low CPU utilisation, as we will explain in the next section.

2.2 Analytical Query Execution Models

Most modern analytical query execution engines use pipelined execution. However, they do not use the tuple-at-a-time iterator model because its design suffers from inefficient CPU utilisation[3]. Two large overheads are the cause of this, which are incurred for every tuple, potentially many times: ① *interpretation overhead*, and ② *function call overhead*. Modern systems have found ways to amortise or overcome both of these overheads.

[3] With inefficient CPU utilisation, we mean a low average number of executed *instructions per cycle* (IPC).

If SQL queries were *static* (*i.e.*, they would never change), developers could implement the corresponding query plans directly in their database system, incorporating the exact query operators, number of columns, and their data types into the plan. Static code allows compilers to generate efficient machine code to evaluate the query plan, which improves CPU utilisation. However, SQL queries are *dynamic*. Database systems must implement generic code that can evaluate any query plan with any number of columns and any data type. Therefore, the code is written to dynamically handle the operators, columns, and types specific to the query plan while evaluating it. Dynamic handling requires *interpreting* and/or *dynamically calling functions* for every tuple, which causes overhead and prevents compiler optimisations, leading to low CPU utilisation.

Query plans may change between queries, but not *during execution*. This is because most database systems strictly enforce a fixed schema for, among other reasons, improved performance. All processed tuples will have the same shape at any given point in a query plan. Therefore, in theory, the overheads do not have to be incurred at such a low level, *i.e.*, for every tuple. Instead, the overheads can be incurred at a higher level to reduce the number of times the overhead is incurred, and modern query execution models have found different ways to do this. Today, most analytical query execution engines use either *vectorisation*, pioneered by VectorWise [18], or *data-centric code-generation*[4], pioneered by HyPer [26]. *Vectorisation* uses a column-oriented layout during execution and processes tuples in batches, amortising per-tuple overheads and allowing operations to efficiently be applied to arrays instead of individual values[5]. *Data-centric code-generation* generates and compiles code specifically for the columns and types required for a query plan[6], eliminating the need for interpretation at runtime, making tuple-at-a-time processing of row-oriented data efficient.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[4] These query engines are referred to as being *vectorised* or *compiled*.

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[5] The size of these batches differs from system to system. The batches in DuckDB's vectorised query engine, for example, contain up to 2,048 tuples.

[6] Compiled query engines can compile the specific shape of a row-oriented tuple into the query plan, *e.g.*, using a `struct` in C++.

Both models have improved their memory access pattern compared to the tuple-at-a-time iterator model. Vectorisation processes small vertical chunks of *CPU cache-resident* vectors at a time and can hide memory latencies by incurring many cache misses simultaneously. Operations on vectors can also often use compiler optimisations such as *loop pipelining* and *single instruction, multiple data* (SIMD) instructions[7], further improving CPU utilisation. Data-centric code-generation does not pull tuples through pipelines from the top down, but instead *pushes* them through a pipeline from the bottom up in a *for-loop* rather than a recursive *call-stack*. This approach allows the compiled query engine to generate efficient machine code that keeps tuples in *CPU registers* for as long as possible. Furthermore, the bottom-up *push-based* loop approach, rather than the top-down *pull-based* recursive approach, also reduces the number of CPU instructions[8], further improving CPU efficiency[9]. The differences between these two models have been researched in-depth [27], and both were found to be efficient, but with different strengths and weaknesses.

[7] Modern CPUs support *SIMD* instructions, where a single instruction operates simultaneously on multiple data points, reducing the number of CPU instructions needed to process data.

[8] The top-down pull-based approach places all control flow in the function call to get the next tuple from an operator, causing some cost to be incurred each time. The bottom-up push-based approach takes the control flow out of this performance-critical code path.

[9] Vectorised query engines may also implement push-based rather than pull-based execution.

[27] Timo Kersten et al. (2018), "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask"

2.3 Frameworks of Parallelism

The parallel evaluation of query plans is another crucial aspect of efficient CPU utilisation, especially on modern many-core CPUs[10]. Non-blocking operators operate on each tuple independently, and are, therefore, trivially parallelisable, which is also referred to as *"embarrassingly parallel"* [28]. Blocking operators operate on the entire input and cannot be trivially parallelised; therefore, a *framework of parallelism* is needed.

[10] CPUs add more cores rather than solely improving single-thread performance because improving single-thread performance faces diminishing returns due to physical limitations like heat and power consumption.

[28] Cleve Moler (1986), "Matrix computation on distributed memory multiprocessors"

An early framework for query parallelisation introduces the *exchange* operator [8], originally proposed for the Volcano system. This *plan-driven* approach to parallelism inserts the exchange operator at strategic places in a query plan and, as its name implies, exchanges data between threads. The exchange operator partitions tuples by, for example, group/join keys and routes these partitions to threads[11], guaranteeing that threads receive entirely disjoint group/join keys. Exchange handles all synchronisation, allowing threads to evaluate *plan fragments*, *i.e.*, parts of the query plan, in a single-threaded fashion, effectively *encapsulating* parallelism. While this approach simplifies the design of operators such as grouped aggregation and joins, it has two shortcomings: ① task scheduling is *coarse-grained* and decided *statically* while planning the query; therefore, skewed data distributions lead to imbalanced workloads, and ② exchanging tuples between threads causes overhead.

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

[11] Note that data can even be routed to different machines for *distributed query processing*.

Morsel-driven parallelism [9] addresses both shortcomings. Instead of partitioning the work by, *e.g.*, group/join keys, morsel-driven parallelism splits the work into *fine-grained* tasks on small fragments of the input data, called *morsels*. These tasks are *dynamically* scheduled while evaluating the query plan. The dynamic fine-grained scheduling allows workloads to be evenly balanced across threads, even if data distributions are skewed, improving parallel CPU utilisation[12].

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[12] Improved *parallel* CPU utilisation refers to the extent to which the available CPU cores are executing useful work concurrently.

Morsel-driven parallelism can achieve near-linear speedups with increased thread counts even when processing skewed data. Tuples are not partitioned and do not need to be rerouted, but threads may encounter the same group/join keys; therefore, operators must be *parallelism-aware*. This complicates their design, as synchronisation, and potentially load-balancing, is needed within blocking operators.

Blocking operators have *four* distinct phases in morsel-driven parallelism[13]. ① *Sink*, in which threads process data from the downstream operator, materialising intermediates in a thread-local state. ② *Combine*, in which threads finish processing their thread-local state, and add intermediates to the thread-global state. ③ *Finalise*, in which the operator is prepared to be used in the next pipeline. If considerable work must be done in this phase, such as inserting tuples into a hash table, it is sometimes necessary to schedule additional operator-specific phase(s) here to parallelise it. ④ *GetData*, in which threads emit data from the operator to the next pipeline. The authors of the morsel-driven parallelism paper [9] show how to parallelise the most important relational operators using this framework.

[13] DuckDB implements morsel-driven parallelism. We use DuckDB's naming convention for these four phases here.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

2.4 External Operator Algorithms

Blocking operators necessitate *materialising* input data. Operators have complete freedom in how to represent the materialised data within the operator[14], as long as the data going in and out of the operator complies with the default of the execution engine. The space requirement of an operator depends on the algorithms and data structures used.

[14] For example, query operators may switch between row- and columnar data orientations.

For example, consider the two most common approaches to join relational data: sort-merge and hashing. With sort-merge, both the inner and outer relations are sorted. Sorting requires full materialisation; therefore, this strategy has a *high space requirement*, namely the sum of the sizes of the inner and outer relations. A simple hash join has two large benefits over sort-merge: ① it has a much lower space requirement, because it only requires materialising the inner relation, as the outer relation can probe the hash table in a streaming fashion, and ② it has a lower theoretical time complexity of $O(n)$ as opposed to the $O(n \log n)$ of sorting, with n the size of the outer relation.

Despite having a high space requirement, sort-merge has the benefit of having a *low memory requirement*. The data can be fully sorted without fitting in memory by using an *external sort* [29], in which small *sorted runs* are generated in main memory before they are merged. Merging the sorted runs and subsequently joining the sorted inner and outer relations uses little memory, as the sorted data can be read sequentially. On the other hand, the simple hash join requires the entire inner relation to fit in main memory. Due to operating in the low memory environments that were available at the time, it is no surprise why traditional disk-based database systems opted for the sort-merge approach.

[29] Donald E. Knuth (1998), *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*

In most database systems, the choice of join algorithm is made prior to query execution. A poor choice has enormous implications, and in some cases, inserting *a single row* into one of the input tables can cause the algorithm choice to change. Choosing a simple hash join can cause queries to abort if the hash table does not fit in memory. Choosing sort-merge can cause execution time to increase by orders of magnitude compared to the case where a hash table would have fit in main memory.

Instead of making a decision *a priori*, database systems should implement query operators to adapt to the data *while executing the query*. By adapting to the data, sort-based query operators can improve their performance, *e.g.*, by reducing the time spent merging, *e.g.*, if some of the data is pre-sorted, and using existing indices if available [30, 31], lowering their space requirement and time complexity. Hash-based query operators can adapt using a *divide-and-conquer* strategy by partitioning the data [32], which reduces their memory requirement. The partitions are typically decided using a few bits of the hash[15]. Not all outer relation tuples have to be materialised for the hash join, as some inner relation tuples can be probed in a streaming fashion [22], which improves performance and also lowers the space requirement. In both sort-based and hash-based operators, adapting to the data results in more robust performance and a more graceful performance degradation as the memory limit is exceeded.

[30] Goetz Graefe (2012), "New Algorithms for Join and Grouping Operations"

[31] Thanh Do et al. (2023), "Efficient Sorting, Duplicate Removal, Grouping, and Aggregation"

[32] Masaru Kitsuregawa et al. (1983), "Application of hash to data base machine and its architecture"

[15] This is referred to as *radix* partitioning.

[22] Leonard D. Shapiro (1986), "Join Processing in Database Systems with Large Main Memories"

2.5 Scalability

When the size of the input data grows relative to the hardware, proportionally less of the data fits higher in the memory hierarchy where access times are low. Therefore, proportionally more data must be stored in and retrieved from lower in the hierarchy where access times are high, which increases query-processing cost[16]. When hardware cannot process a large data set (fast enough) anymore, there are *two* options: ① *scaling up* (also referred to as *vertical scaling*), which means increasing a single machine's resources such as memory and storage, and ② *scaling out* (also referred to as *horizontal scaling*), which means increasing the number of machines (nodes) contributing to the completion of the workload.

[16] See Figure 1.1 for an illustration of the memory hierarchy.

One can only add so many resources to a single computer; therefore, vertical scaling is limited. Horizontal scaling, on the other hand, has no real limit and is the only way to complete some workloads. Scaling out requires distributed software applications, which brings many challenges. To this end, the *MapReduce* [33] programming model was invented, in which developers only have to implement a `map` and `reduce` function. The underlying runtime will then automatically parallelise the computation across large-scale clusters. *Spark* [34], MapReduce's successor, improves performance by orders of magnitude by keeping data in memory.

[33] Jeffrey Dean et al. (2004), "MapReduce: Simplified Data Processing on Large Clusters"

[34] Matei Zaharia et al. (2012), "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing"

Despite the success of MapReduce and Spark, especially during the peak of the *Big Data hype*, there are many problems with distributed data processing. Much like increasing the number of cores in a single machine, one cannot expect performance to improve linearly by adding more compute nodes. More specifically, distributed data processing may add overheads that would not be present in single-node implementations, which makes scaling out unattractive for many problems [35]. Furthermore, the MapReduce programming model requires developers to implement imperative Java code for every job, and incurs performance problems due to its design that existing parallel DBMSes already solved [36]. Many MapReduce jobs could have been realised by implementing `map` and `reduce` as user-defined functions (UDFs) in a declarative language like SQL, which would have allowed for easier additional data manipulation.

[35] Frank McSherry et al. (2015), "Scalability! but at what cost?"

[36] Michael Stonebraker et al. (2024), "What Goes Around Comes Around... And Around..."

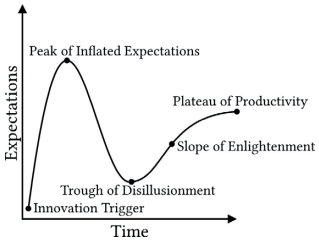

Figure 2.2 The Gartner Hype Cycle.

As the Big Data hype moved past the "Peak of Inflated Expectations" in the *Gartner Hype Cycle* [37], depicted in Figure 2.2, people have realised that most applications do not need distributed data systems. Some have even declared Big Data to be "dead" [38]. The low demand for Big Data solutions became even more apparent after Amazon Redshift and Snowflake reported how their users use their systems [39, 40]. Specifically in Redshift, only 0.03% of queries in the data set, or 3 in 10,000, query more than 10 TB of data [41]. While the amount of data that users store may be much larger, analysis is usually only done on a small portion of the data at a time. Modern database systems and file formats like Parquet [42] allow for optimisations such as partition pruning, column projection, filter pushdown, segment selection, and more. As a result, systems often read only a fraction of each table [39]. Even if a lot of data is read, the data is often immediately reduced, *e.g.*, by an aggregation. Most queries can be processed by a single node, even if the data is distributed across multiple nodes.

[37] Jackie Fenn et al. (2018), *Understanding Gartner's Hype Cycles*

[38] Jordan Tigani (2023), *Big Data Is Dead*

[39] Jordan Tigani (2024), *Redshift Files: The Hunt for Big Data*

[40] George Fraser (2024), *How do people use Snowflake and Redshift?*

[41] Alexander van Renen et al. (2024), "Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet"

[42] Deepak Vohra (2016), "Apache Parquet"

[39] Jordan Tigani (2024), *Redshift Files: The Hunt for Big Data*

2.6 DuckDB

DuckDB is an open-source column-oriented relational database management system[17]. It was originally developed by Mark Raasveldt and Hannes Mühleisen in the Database Architectures group at Centrum Wiskunde & Informatica in Amsterdam, where vectorised query execution [18] was initially invented. It is used in-process, rather than with a traditional client-server model, which makes it easier to use and eliminates client-server serialisation overhead [43]. DuckDB implements a push-based vectorised query execution engine and parallelises query execution using morsel-driven parallelism [9]. It is designed to be used on a single node, *i.e.*, for scale-up architectures. All of the research carried out for this thesis has been implemented in DuckDB [16] and is used by millions of users worldwide.

[17] Information about DuckDB is available at `https://duckdb.org/`, and the source code is available at `https://github.com/duckdb/duckdb`.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[43] Mark Raasveldt et al. (2017), "Don't Hold My Data Hostage: A Case for Client Protocol Redesign"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

3. Data Orientation

Now that we have the required background knowledge, we dive into the design of physical query operators. In this chapter, we research the effects of data orientation on the efficiency of sorting relational data. In a system like DuckDB, which uses a columnar data orientation during query execution, using a row-oriented data layout requires converting between row- and column-oriented layouts, as shown in Figure 3.1.

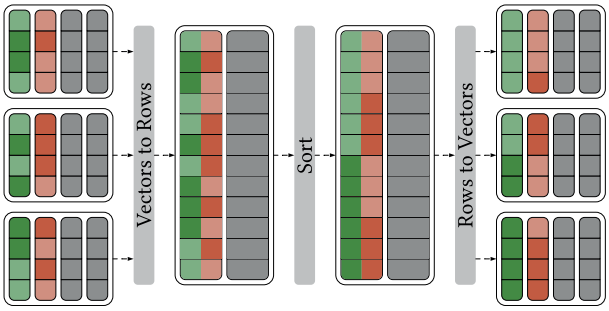

Figure 3.1 Converting vectors to rows, sorting them, and converting them back to vectors. The columns that appear in the order clause are coloured, and the other selected columns are in grey.

3.1 Introduction

Sorting is one of the most well-studied problems in computer science. Research on efficient sorting algorithms focuses on crucial issues such as cache-efficiency [44], reducing branch mispredictions [45], parallelism [46], and worst-case patterns [47], but is mostly limited to sorting large arrays of integers. Database systems research focuses on many of the same issues [18, 9, 48], and database systems have some of the most practical use-cases of sorting: the `ORDER BY` and `WINDOW` operators explicitly invoke sorting, but other operations such as index construction, merge joins, and inequality joins [49] may implicitly rely on sorting. Therefore, an efficient sort implementation is crucial for providing fast query response times, especially for analytical data management systems. However, sorting relational data is more involved than sorting an array of integers. Relatively less research has gone into sorting relational data efficiently, especially compared to research on other operators, such as joins [50].

Although relational sorting implementations benefit from advances in general-purpose sorting algorithms, merely using such an algorithm will not make a relational sorting implementation efficient by default, as we will demonstrate in this chapter. Two operations dominate the cost of sorting: comparing and moving tuples [51]. These two operations are more complex for relational data than for integers. The comparison function that arises from the `ORDER BY` clause can be arbitrarily complex and contain any type the system supports. The tuples that are then moved can also have any type the database system supports, and there can be an arbitrary number of columns. Inefficient implementations of these operations will lead to inefficient sorting, even if the sorting algorithm is efficient.

[44] Anthony LaMarca et al. (1999), "The Influence of Caches on the Performance of Sorting"

[45] Stefan Edelkamp et al. (2019), "BlockQuicksort: Avoiding Branch Mispredictions in Quicksort"

[46] Oded Green et al. (2014), "Merge Path - A Visually Intuitive Approach to Parallel Merging"

[47] Orson R. L. Peters (2021), "Pattern-defeating Quicksort"

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[48] Michael Freitag et al. (2020), "Adopting Worst-Case Optimal Joins in Relational Database Systems"

[49] Zuhair Khayyat et al. (2015), "Lightning Fast and Space Efficient Inequality Joins"

[50] Maximilian Bandle et al. (2021), "To Partition, or Not to Partition, That is the Join Question in a Real System"

[51] Goetz Graefe (2006), "Implementing Sorting in Database Systems"

A system's query execution engine affects how these operations can be implemented and the efficiency of these implementations. The engines of most modern OLAP systems are based on either vectorisation, pioneered by VectorWise [18], or data-centric code-generation, pioneered by HyPer [26]. The strengths and weaknesses of these two query execution models have not been researched in the context of relational sorting.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

Contributions. As stated in Chapter 1, the main contributions of this chapter are the following:

- We present a comprehensive experimental evaluation of how data orientation affects relational sorting performance. We show that a row-oriented layout is more efficient, regardless of whether systems use vectorisation or code generation.
- We measure how interpretation overhead affects comparisons, which is one of the main costs of sorting. We show that compiled query engines are efficient, while interpreted engines suffer from various overheads. We then show that these overheads can be overcome with existing techniques.
- We propose to use *pointer swizzling*[1], rather than full (de-)serialisation, to spill in-memory data with pointers to storage. We then read the data back into memory, potentially to a different address, without invalidating the pointers. This allows what is essentially an in-memory merge sort implementation to sort data that is larger than memory.

[1] *Pointer swizzling* is the in-place conversion of references based on name or position into direct pointer references.

We have implemented the proposed techniques in DuckDB [16]. They have been available since the 0.3.0 release.

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

Outline. We discuss the challenges of implementing relational sorting in Section 3.2 (page 25), and describe our methodology in Section 3.3 (page 27). In Section 3.4 (page 29), we evaluate several approaches to sorting relational data in row- and column-oriented layouts. Then, in Section 3.5 (page 33), we discuss how the query execution engine affects sorting performance. We demonstrate that the overheads that vectorised interpreted engines face can be overcome with several existing techniques in Section 3.6 (page 36). We implement a row-oriented sort that includes these techniques within DuckDB. In Section 3.7 (page 39), we compare our implementation with four high-performance analytical database systems: ClickHouse [52], MonetDB [53], HyPer [26], and Umbra [54]. The results of this comparison show that DuckDB's interpreted row-oriented sort implementation outperforms ClickHouse's and MonetDB's sort implementations, which use a column-oriented layout, and matches or outperforms HyPer's and Umbra's compiled row-oriented sort implementations. Before summarising the chapter in Section 3.9 (page 47), we explain how columns that are projected but are not part of the `ORDER BY` clause can be handled for larger-than-memory sorting in Section 3.8 (page 45).

[52] Baktagul Imasheva et al. (2019), "The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse"

[53] Stratos Idreos et al. (2012), "MonetDB: Two Decades of Research in Column-oriented Database Architectures"

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

3.2 Sorting Relational Data

Database systems use sorting for many purposes [51]. Sorting can be invoked explicitly using SQL, but also implicitly for many purposes such as join algorithms [49], improving run-length encoding compression [55], and zone map [56] effectiveness. Because it is widely applicable, any database system needs an efficient sorting implementation, especially OLAP systems that aim to have low query response times. However, efficiently sorting relational data is more complex than efficiently sorting an array of integers, which is the default benchmark for sorting algorithms.

[51] Goetz Graefe (2006), "Implementing Sorting in Database Systems"

[49] Zuhair Khayyat et al. (2015), "Lightning Fast and Space Efficient Inequality Joins"

[55] Daniel Lemire et al. (2011), "Reordering Columns for Smaller Indexes"

[56] Guido Moerkotte (1998), "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing"

Simple Query, Complex Comparison. A relational sorting implementation must be able to sort all projected columns by one or more predicates, whether sorting is invoked explicitly in an `ORDER BY` clause or implicitly by other means. Take the query in Listing 3.1, for example. This query specifies which columns to return and how they should be sorted, *i.e.*, it describes how rows should be compared to produce the ordered query result. All columns from the `customer` table should be returned, sorted by the column `c_birth_country` in descending order, with `NULL` values coming last. Where rows have the same value in that column, they should be sorted by `c_birth_year` in ascending order with `NULL` values coming first. We will refer to the columns that appear in the `ORDER BY` clause as *key* columns and all other selected columns as *payload* columns, as is common when discussing other relational algorithms such as joins.

Listing 3.1 An example query which sorts the data according to the `ORDER BY` clause.

```
SELECT *
FROM customer
ORDER BY
  c_birth_country DESC NULLS LAST,
  c_birth_year ASC NULLS FIRST;
```

Even this *two-key query* already requires a complex row comparison. In contrast, comparing integers is done using a single instruction. For tuples, a straightforward comparator implementation will lead to code with many branches that degrade performance. Because the comparator is frequently used during sorting, it should be implemented as efficiently as possible. For example, quicksort can be sped up by reducing branches [45], but this is only effective if the comparison function is branchless.

[45] Stefan Edelkamp et al. (2019), "BlockQuicksort: Avoiding Branch Mispredictions in Quicksort"

Physical Data Representation. How a relational sort implementation physically represents data in memory also affects comparison efficiency. If the keys belonging to a single row are not stored consecutively in memory, which is the case for a column-oriented layout, comparing tuples causes random access for each compared value, which may cause cache misses. Improving cache locality speeds up many sorting algorithms [44].

[44] Anthony LaMarca et al. (1999), "The Influence of Caches on the Performance of Sorting"

Besides determining the order in which to return the tuples by comparing and sorting them, the data must also be physically reordered. Reordering can be done early, while sorting, or late, right before the data is output to the downstream operator. This is trivial for an array of integers: we sort the array with a general-purpose sorting algorithm, which results in the data being in the correct order. It

is less obvious for relational data because there are many ways to represent tuples. Sorting conceptually operates on tuples as units, but systems with a vectorised engine use a column-oriented layout during execution. For such systems, it might be beneficial to convert the data to a row-oriented layout, also called the N-ary Storage Model (NSM), and then convert it back to a column-oriented layout, also called the Decomposition Storage Model (DSM) [57], after completing the sort. Vectorised engines such as VectorWise [18] already do this for hash tables in joins and aggregations to improve cache efficiency [19].

[57] Anastassia Ailamaki et al. (2002), "Data Page Layouts for Relational Databases on Deep Memory Hierarchies"

[19] Marcin Zukowski et al. (2008), "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing"

Additionally, we can process the key and payload columns separately, retrieving the payload in the correct order after sorting the key columns. The question is then whether it is better to convert either, neither, or both to NSM. This conversion is a trade-off between cache locality and data movement. Converting from DSM to NSM results in a better memory access pattern when retrieving the payload in the correct order because values that belong together are close together in memory. This conversion, however, requires moving all the data from a column- to a row-oriented layout. Payload handling is an essential part of relational sorting. Although we focus on sorting key columns in this chapter, we explain how we implemented payload handling in DuckDB in Section 3.8 (page 45).

Parallelism. As we will see in Section 3.7 (page 39), when we discuss the systems under benchmark, many database systems parallelise sorting by generating sorted runs with thread-local sorts[2], *e.g.*, with morsel-driven parallelism [9]. This step is followed by a parallel merge sort, which fully merges the runs to sort the data.

[2] Also referred to as *run generation*.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

Comparison-based sorting algorithms, such as quicksort and merge sort, use on average $O(n \cdot \log n)$ comparisons for an input size of n. If we generate k sorted runs, the average number of comparisons performed during run generation is

$$\text{comp}_A = k \cdot \frac{n}{k} \cdot \log\left(\frac{n}{k}\right) = n \cdot \log n - n \cdot \log k,$$

as we sort k runs of size $\frac{n}{k}$. The average number of comparisons performed during merge sort is $\text{comp}_B = n \cdot \log k$, as we have to do $\log k$ comparisons to find the smallest value out of k runs, n times. To find the point at which we have more comparisons during merge sort, we can solve the inequality $\text{comp}_A > \text{comp}_B$. The solution to this inequality is $k > \sqrt{n}$. If the data fits in memory, each thread will generally generate one sorted run; therefore, the number of generated runs k equals the number of threads. The input size n can be arbitrarily large; therefore, comp_A is almost always larger than comp_B. For example, for $n = 1{,}000{,}000$ and $k = 16$, approx. 80% of the total number of comparisons are performed during run generation, on average. Therefore, this chapter focuses on run generation, as run generation takes up most of the time spent for many applications of relational sorting.

It is unclear what the most efficient approach will be for a given system without benchmarking different sorting implementations. Differences in execution engine, hardware characteristics like CPU cache size, and functionality requirements all affect how to design a relational sorting method. However, implementing a relational operator such as sort in a database system requires substantial engineering effort, let alone implementing multiple variants to determine the best-performing one. Therefore, we use micro-benchmarks to evaluate several approaches for sorting relational data to determine their effectiveness.

3.3 Methodology & Micro-Benchmarks

To isolate the fundamental properties of sorting row- and column-oriented data, as well as the fundamental properties of sorting in compiled and vectorised interpreted query execution engines, we implement micro-benchmarks. By using micro-benchmarks, we do not measure the time it takes to parse and optimise queries and the overhead of result set serialisation [43], which will differ across database systems. We implement several approaches to sorting relational data in C++. All of the approaches use `std::sort`, an introspective [3] sort [58] implementation. The `std::sort` algorithm is not state-of-the-art, as it has already been improved upon in various ways [45, 47]. However, we wish to measure the effect of different data layouts, tuple comparison methods, and execution engines, regardless of the sorting algorithm. We compare `std::sort` only against itself and not against other sorting algorithms so that we can isolate the effects of the data layouts, comparison methods, and execution engines. To verify whether these effects generalise to other sorting algorithms, we replicate our experiments with `std::stable_sort`, which is a stable sorting algorithm based on merge sort. Merge sort has a different cache behaviour from quicksort, as merge sort uses primarily sequential data access. Again, we only compare `std::stable_sort` against itself, not against other sorting algorithms.

[43] Mark Raasveldt et al. (2017), "Don't Hold My Data Hostage: A Case for Client Protocol Redesign"

[3] An *introspective* sort is a hybrid sorting algorithm that provides both fast average performance and (asymptotically) optimal worst-case performance.

[58] David R. Musser (1997), "Introspective Sorting and Selection Algorithms"

[45] Stefan Edelkamp et al. (2019), "BlockQuicksort: Avoiding Branch Mispredictions in Quicksort"

[47] Orson R. L. Peters (2021), "Pattern-defeating Quicksort"

Micro-Benchmark Workload. The data for our micro-benchmarks consists of columns of unsigned 32-bit integers. These are generated by sampling from two distributions[4]:

Random	Random uniform
CorrelatedP	Correlated with 128 unique values, with P the correlation probability

We vary the number of key columns from 1 to 4 and the number of rows from 2^{12} to 2^{24}[5]. The Random data distribution has virtually no duplicate values in each column. The CorrelatedP distribution has 128 unique values per column, and the first column is distributed uniformly. For subsequent columns, P gives the probability of a column $C+1$ correlating with column C. For example, in the Correlated0.5 distribution, when we compare two tuples with an equal value in

[4] The source code for our micro-benchmarks can be found at `https://github.com/lnkuiper/experiments/tree/master/sorting_simulation`, archived at `https://doi.org/10.5281/zenodo.15967009`

[5] Note that although we use only integers, like most sorting algorithm research, we create a more complex comparison function by varying the number of key columns, and changing the data orientation. We evaluate string sorting performance in Section 3.7 (page 39).

column C, there is a 50% probability that their values in column $C+1$ will also be equal. Duplicate values in the key columns lead to ties when comparing values during sorting, which requires the comparison function to compare the next key column as well. Therefore, depending on the correlation, more comparisons need to be performed. For high correlations, tuples are more likely to have the exact same values in all key columns, reducing the total number of unique tuples.

As we will demonstrate in the following, memory access patterns, branch mispredictions, and interpretation overhead are the main differentiating factors in the performance of sorting implementations. The memory access patterns of sorting row- and column-oriented data we measure in our micro-benchmarks are the same regardless of data type. The same holds for branch mispredictions and interpretation overhead. We purposely keep our micro-benchmarks simple, but we include experiments with other data types in our end-to-end benchmarks in Section 3.7 (page 39).

Experimental Setup. To measure the runtime of our micro-benchmarks, we use the `m5d.metal` AWS EC2 instance unless noted otherwise. This instance has an Intel Xeon Platinum 8259CL CPU with 48 cores (96 threads) and 384 GB of RAM. We use Ubuntu 20.04 as the OS and compile our code with Clang 12.0. We repeat each experiment five times and report only the median runtime.

We use CPU performance counters to analyse and understand the properties of different approaches further. Measuring these counters is impossible on most AWS EC2 instances due to the restrictions of virtual machines. The `metal` instance does allow reading the performance counters, which is why we have chosen to use it. We obtain performance counters using Linux's `perf` tool[6]. We measure the number of branch mispredictions, and L1 data cache load misses with `-e branch-misses,L1-dcache-load-misses`. We run the performance counter measurements just once.

In Section 3.7 (page 39), we run end-to-end runtime benchmarks to compare the performance of full-fledged database systems. Because we are not measuring performance counters, we do not need to use the `metal` instance, and we use the smaller `m5d.8xlarge` instance instead. This instance has exactly the same hardware as the `m5d.metal` instance, but it is a virtual machine that uses one-third of its resources.

All instances used in our experiments have SSDs. Detailed hardware specifications are listed in Table 3.1.

[6] `perf` is a Linux performance analysis tool that provides low-overhead access to hardware performance counters and kernel tracepoints, enabling fine-grained profiling of CPU behavior, cache usage, and kernel/user-space execution. It is widely used to identify performance bottlenecks at the system and application level.

Table 3.1 Specification of hardware used in experiments.

Instance	`m5d.metal`	`m5d.8xlarge`
CPU Brand	Intel	Intel
CPU Model	8259CL	8259CL
Architecture	x86	x86
Cores	48	16
Threads	96	32
Clock Rate	2.5-3.5 GHz	2.5-3.5 GHz
L1 Cache	32 KB	32 KB
RAM	384 GB	128 GB

3.4 DSM vs. NSM

In this section, we experimentally evaluate the efficiency and performance characteristics of sorting relational data in row- and column-oriented layouts. We sort only the key columns because we can retrieve the payload in the correct order after sorting the keys to create a fully sorted run. To collect the payload, we need to track which keys correspond to which payload, *e.g.*, using a pointer or a row ID. Furthermore, we assume that all input has been materialised, *i.e.*, the sort operator has collected all input data already in a row or column-oriented layout, allowing us to isolate raw sorting performance further.

There are two obvious ways of comparing tuples of relational data when there are multiple key columns:

tuple-at-a-time Iterate through the key columns with each comparison. Compare values until we find one that is not equal or until we have iterated through all columns in the `ORDER BY` clause.

subsort Sort everything by the first key column, then identify the tuples with an equal value in this column, and sort these by the second key column. Repeat until all key columns are sorted.

We apply these approaches to both layouts.

Sorting Column-Oriented Data. A column-oriented layout necessitates sorting row indices rather than sorting the data in the columns directly because we need to use the indices to access the data in the columns. After sorting, the same sorted indices are used to retrieve the payload in the correct order. An example implementation of this is shown in Listing 3.2.

Listing 3.2 An example implementation of the *tuple-at-a-time* approach for a column-oriented layout using `std::sort` in `C++`.

```
// Compare two tuples
// using their row ID
bool compare(l, r, cols) {
  size_t i = 0;
  for (; i < cols.size() - 1; i++) {
    if (cols[i][l] != cols[i][r])
      break;
  }
  return cols[i][l] < cols[i][r];
}
// Sort N row indices
// by the key column values
std::sort(idxs, idxs + N,
  [&] (size_t l, size_t r) {
    return compare(l, r, cols);
  }
);
```

In the example implementation, tuples are represented by their indices `idxs` and compared using their respective values in `cols`. The *subsort* implementation works similarly: it also sorts by indices but has a less complex comparison function that only compares one column. Additionally, the *subsort* approach has to identify tied tuples after each pass and recurse until there are no more ties or until it has passed over all columns.

The *tuple-at-a-time* sorting approach for the column-oriented layout has three major drawbacks. (1) Comparing two tuples causes multiple random accesses with each comparison. If the tuples have the same value in the first key column, we need to compare their value in the second key column, and so forth. The more duplicate values there are, the more random access is required. Therefore, this approach suffers from data distributions with many duplicates and skewed data distributions. (2) The comparison function has branches, namely, whether to compare the values in the next key column if the values in the current key column are equal. Again, this may be difficult to predict depending on the distribution: if there are no duplicates, the branch predictor will correctly predict that it does not need to

compare the next column. ③ Sorting columnar data sorts indices rather than the data in the key columns itself, *i.e.*, the data in the key columns never actually moves. Sorting algorithms generally move similar values next to each other in memory, which improves cache locality. By not physically moving the data, the columnar approach benefits less from the cache efficiency of a sorting algorithm.

The *subsort* approach improves on this by only causing random access in one column at a time, mitigating drawback ①, and comparing a single column at a time and therefore having no branches in the comparison function, eliminating drawback ②. Furthermore, sorting one column at a time reduces the cache pressure because only a single memory region (albeit large) is accessed at a time, which mitigates drawback ③ slightly.

We measure performance counters of both methods in our micro-benchmark and show the results in Table 3.2. There are almost no duplicates in the Random data distribution, and the second column rarely has to be randomly accessed when comparing tuples. Therefore, both approaches operate almost exactly the same and sort the data by only accessing the first column. This explains why the number of cache misses and branch mispredictions is similar. For the correlated data distributions, the *subsort* approach incurs fewer cache misses and branch mispredictions as expected.

Table 3.2 L1 *cache misses* (**CM**) and *branch mispredictions* (**BM**) (both in billions, lower is better) of sorting 2^{24} rows of 4 key columns in *column-oriented* (**C**) layout, Correlatedo.5 distribution, with the *tuple-at-a-time* (T) and *subsort* (S) approaches, with `std::sort`.

	CM		BM	
Distribution	C/T	C/S	C/T	C/S
Random	**0.41**	0.45	**0.17**	**0.17**
Correlatedo.1	1.48	**0.96**	0.24	**0.19**
Correlatedo.5	1.60	**1.05**	0.20	**0.16**
Correlatedo.9	1.46	**1.11**	0.11	**0.08**

CPU performance is affected negatively by cache misses and branch mispredictions, which should translate into a worse overall sorting performance. We measure the runtime of both approaches using our micro-benchmark and show the relative runtime of *subsort* compared to the *tuple-at-a-time* approach in Figure 3.2. A relative runtime of 2.00 means that the *subsort* approach is twice as fast as the *tuple-at-a-time* approach, *i.e.*, it finishes sorting in half the time.

When there is only one key column, the approaches are virtually equal. As expected, the relative runtime is close to 1 for all inputs for the Random data distribution because there are almost no duplicates. For the Correlated distributions, the *subsort* approach is better the more rows and the more key columns there are. With more rows and key columns, the *tuple-at-a-time* approach performs relatively worse because the columns no longer fit in the CPU's cache. The increased cache pressure hurts its performance.

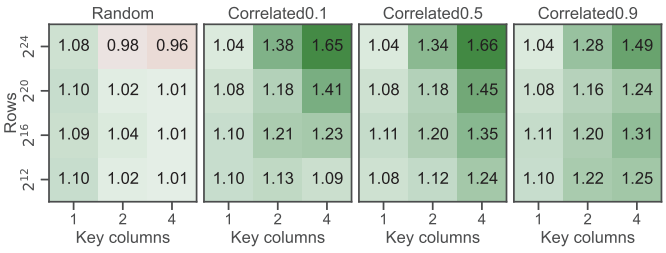

Figure 3.2 Relative runtime (higher is better) of the *subsort* approach compared to the *tuple-at-a-time* approach on a column-oriented layout with **`std::sort`**.

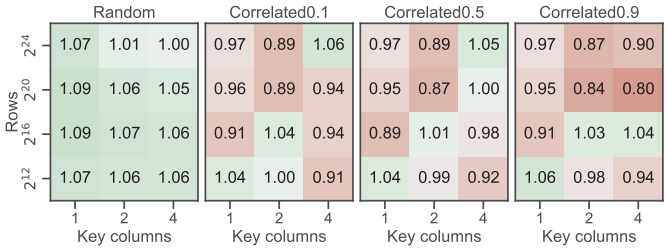

Figure 3.3 Relative runtime (higher is better) of the *subsort* approach compared to the *tuple-at-a-time* approach on a column-oriented layout with `std::stable_sort`.

Using `std::stable_sort`, we replicate this benchmark to verify that our findings generalise to other sorting algorithms. We show the results in Figure 3.3. With this sorting algorithm, the approaches are much closer, and *subsort* is often slightly slower than *tuple-at-a-time*. The `std::stable_sort` implementation is based on merge sort, which mostly does sequential access. Therefore, the worse cache behaviour of the *tuple-at-a-time* comparison approach does not hurt its performance as much, comparatively. Based on these findings, we cannot conclude that the *subsort* approach is better than the *tuple-at-a-time* approach on column-oriented data, as the results depend on the sorting algorithm. In the next paragraph, we will see how the different comparison strategies will perform on row-oriented data.

Sorting Row-Oriented Data. Unlike with column-oriented data, we do not have to use row indices when sorting row-oriented data. We address rows directly because all values that we compare are co-located. The row indices (or pointers) are still needed to retrieve the payload in the correct order after sorting. We pack these within the row, which, for example, can be achieved with a struct in C++, as shown in Listing 3.3.

Listing 3.3 Example of how row-oriented data can be represented as a `struct` in C++.

```
struct OrderKey {
  // Key columns
  uint32_t col1;
  uint32_t col2;
  // Index (or pointer) to payload
  size_t idx;
};
```

The row-oriented data is sorted by calling, *e.g.*, `std::sort` on an array of the `OrderKey` structs. The comparison function used to sort is then defined as a comparison function between the key column values in these structs.

Like the column-oriented data, we sort the row-oriented data with the *tuple-at-a-time* approach. We sort rows with the *subsort* approach as well, although, at first glance, it is less intuitive to do this with rows. Like before, *subsort* simplifies the comparison function while sorting, as it does not need branches. The implementation of *subsort* for rows is the same as for column-oriented data: we sort, identify which tuples are tied, sort these by the next column, until all sort keys have been processed.

We measure performance counters of both methods in our micro-benchmark and show the results in Table 3.3. When we compare this with Table 3.2, sorting row-oriented data incurs an *order of magnitude fewer* cache misses. The number of branch misses is almost exactly the same, as expected, since the layout does not affect the comparisons to be done with the same sorting algorithm. The *subsort* approach incurs

Table 3.3 L1 *cache misses* (**CM**) and *branch mispredictions* (**BM**) (both in billions, lower is better) of sorting 2^{24} rows of 4 key columns in **row-oriented (R) data layout**, Correlated0.5 distribution, with the *tuple-at-a-time* (T), and *subsort* (S) approaches, with `std::sort`.

	CM		BM	
Distribution	R/T	R/S	R/T	R/S
Random	**0.10**	**0.10**	0.18	**0.17**
Correlated0.1	**0.10**	0.17	0.25	**0.20**
Correlated0.5	**0.10**	0.19	0.21	**0.17**
Correlated0.9	**0.09**	0.22	0.11	**0.08**

fewer branch mispredictions than the *tuple-at-a-time* approach on every data distribution because there are no branches in the comparison function. The *subsort* approach incurs slightly more cache misses than *tuple-at-a-time*, due to re-scanning the data for tied tuples after each sort.

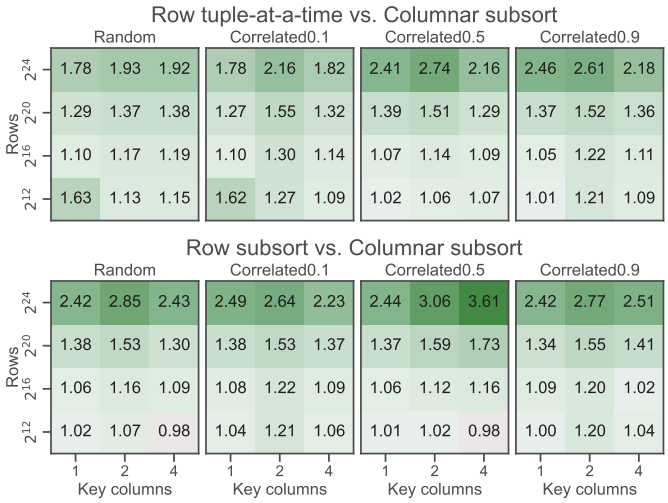

Figure 3.4 Relative runtime (higher is better) of the *tuple-at-a-time* and *subsort* approaches on row-oriented layout compared to the *subsort* approach on a column-oriented layout, with `std::sort`.

We measure the runtime of the row-oriented approaches and show the results in Figure 3.4. We use the columnar *subsort* approach from earlier as a baseline. The results show that sorting row-oriented data is more efficient than sorting column-oriented data, as the relative runtime of the row sorting approaches is greater than 1 for almost all inputs. For smaller input sizes, the data fits in the CPU's cache; therefore, random access is not an issue for the column- and row-oriented layouts. For larger input sizes, the data does not fit in the CPU's cache, and the row-oriented layout's improved cache locality results in better performance. Like with the column-oriented layout, the *subsort* approach is faster for most inputs than the *tuple-at-a-time* approach.

Like before, we replicate this exact experiment with `std::stable_sort` and show the results in Figure 3.5. When we compare this to Figure 3.4, we can see that the results are similar. However, the relative runtimes are slightly better for `std::sort`, so the benefit of the row-oriented layout is higher for this algorithm than for `std::stable_sort`. The *subsort* approach performs better than *tuple-at-a-time* with `std::stable_sort` on the row-oriented layout, which was not the case for the column-oriented layout.

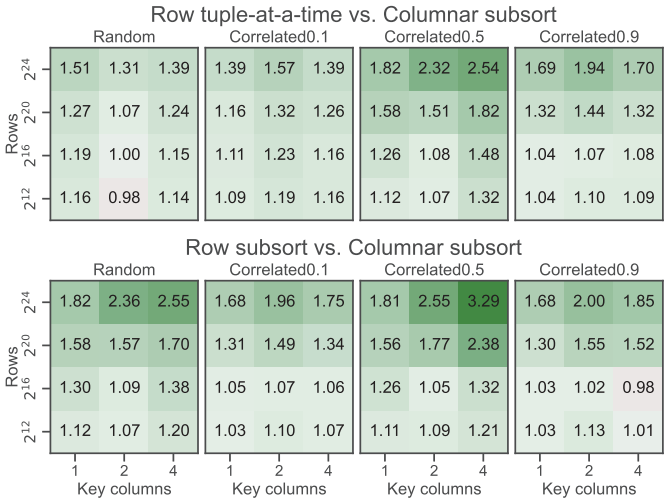

Figure 3.5 Relative runtime (higher is better) of the *tuple-at-a-time* and *subsort* approaches on row-oriented data compared to the *subsort* approach on a column-oriented data, with `std::stable_sort`.

In summary, NSM (row-oriented) tuple representation outperforms DSM (column-oriented) when sorting relational data. This is true *regardless of data distribution*, and the performance gain is noticeable when sorting a large number of tuples. In a columnar execution engine, using a row-oriented layout for the sorting operator requires converting the input to rows, performing the actual sort, and then converting the output to a column-oriented layout again. It remains to be seen whether the performance benefits of using a row-oriented layout for sorting are still apparent, considering this additional cost. This will be evaluated in end-to-end benchmarks in Section 3.7 (page 39).

3.5 Query Engines and Sorting

The Volcano [25] tuple-at-a-time model for pipelined processing causes high interpretation overhead, making it unsuitable for OLAP systems. Two main approaches to address this overhead have emerged: Vectorised query execution [18] amortises this overhead with vector-at-a-time query execution. Compiled query execution [26] uses just-in-time (JIT) compilation to generate code specialised for the operators and types present in the query, eliminating interpretation overhead. These two different models have a similar performance in OLAP workloads [27]. Although the concepts of vectorisation and compilation are not strictly orthogonal [59], most systems adopt one of the two models.

In a query execution pipeline, sorting is a blocking operator: it must consume all input data before outputting any because the last input row could be the first output row. This necessitates materialising the input data. To a certain extent, operators that materialise their input have more 'freedom' than streaming operators. They can internally represent the data in any way they see fit as long as they output

[25] G. Graefe (1994), "Volcano-An Extensible and Parallel Query Evaluation System"

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[27] Timo Kersten et al. (2018), "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask"

[59] Juliusz Sompolski et al. (2011), "Vectorization vs. Compilation in Query Execution"

their data in a way that is compliant with what the execution engine expects. Although converting the data from and to a different layout comes with a cost, it can speed up query processing [60, 19] because it results in better memory access patterns.

[60] Yihong Zhao et al. (1997), "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates"

[19] Marcin Zukowski et al. (2008), "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing"

Both vectorised and compiled query engines have to choose how to materialise input data for the sort operator. As we saw in the previous section, the data layout affects sorting efficiency because it affects the cost of comparing and moving data. However, within a database system, sorting is used for many purposes. How other operators use sorted data internally, such as merge- and inequality joins [49], should also be taken into account. In this section, we discuss the effect that the different query execution engines have on sorting.

[49] Zuhair Khayyat et al. (2015), "Lightning Fast and Space Efficient Inequality Joins"

Compiled Sorting. The previous section showed that sorting a row-oriented layout is more efficient than sorting a column-oriented layout. This is especially true when we sort many tuples by many key columns. Compiling query engines use tuple-at-a-time processing on generated data types [27], such as the `OrderKey` struct that we used in our micro-benchmarks. An array of such structs is essentially relational data in row-oriented layout. This layout allows compiling query engines to sort using an efficient iterator, such as the C++ iterator interface that is used for `std::sort` and other efficient sort implementations [45, 47]. Furthermore, compiling engines can also compile the comparison function itself, which removes interpretation and function call overhead.

[27] Timo Kersten et al. (2018), "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask"

[45] Stefan Edelkamp et al. (2019), "Block-Quicksort: Avoiding Branch Mispredictions in Quicksort"

[47] Orson R. L. Peters (2021), "Pattern-defeating Quicksort"

In short, compiling query engines can use efficient compiled operations to compare and move values, which are the two main costs of sorting. Compiling query engines can adopt the *tuple-at-a-time* or *subsort* approaches and achieve sorting performance that matches the best results observed in our micro-benchmarks so far. However, as elegant as they seem, compiled query engines require complex compiler infrastructure to generate the code for each query [26], which increases system complexity and reduces debuggability [27].

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[27] Timo Kersten et al. (2018), "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask"

Vectorised (Interpreted) Sorting. Vectorisation amortises the interpretation and function call overhead of tuple-at-a-time processing and does not require compilation. This approach yields excellent performance in many cases because many relational operations are vectorisable. The same is true for sorting: in a system with a vectorised interpreted engine, the *subsort* approach can be used to interpret the type and `ASC/DESC`, `NULLS FIRST/LAST` order only once per key column. However, as we have seen, the columnar *subsort* approach is much less efficient than sorting row data for large input sizes.

Furthermore, some operations in database systems cannot use the *subsort* approach. Examples of this are merge sort, merge joins, and inequality joins [49]. These operations iterate sequentially over sorted runs and compare tuples. For example, in a 2-way merge sort, we iterate over a left and a right sorted run. When merging the runs, the decision of incrementing either the left or right iterator relies on a full tuple comparison, *i.e.*, comparing all key columns. Fully comparing tuples in an interpreted engine requires interpreting a type and a sort order for each key column or performing a function callback for each key column in the tuple comparison. The former creates interpretation overhead, and the latter creates function call overhead in the comparison function. This overhead is costly [18] and decreases CPU efficiency, especially when called for every tuple, which is necessary for comparing tuples of sorted data.

[49] Zuhair Khayyat et al. (2015), "Lightning Fast and Space Efficient Inequality Joins"

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

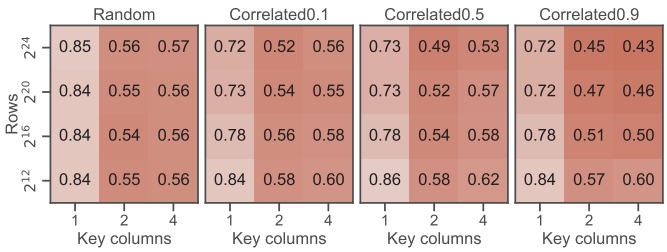

Figure 3.6 Relative runtime (higher is better) of a *tuple-at-a-time* approach with a dynamic comparator compared to a static *tuple-at-a-time* comparator on data in row-oriented layout, with `std::sort`.

We illustrate this cost by comparing two *tuple-at-a-time* approaches to sorting the row-oriented data in our micro-benchmark. The approaches are identical except that one has a statically compiled comparison function, and the other uses a dynamic function call each time it compares values, incurring function call overhead on every comparison. We show the results of this experiment in Figure 3.6. As expected, a dynamic function call is always slower than a statically compiled comparison by roughly a factor of 2. This effect is more substantial with more key columns. This benchmark simplifies what would be implemented in a full-fledged database system with a vectorised interpreted engine. However, the experiment illustrates the kind of overhead that interpreted systems face.

In our experiments in the previous section, we showed that sorting data in a row-oriented layout is more efficient than sorting data in a column-oriented layout. In this section, we showed that vectorised interpreted execution engines suffer from overheads that hurt sorting efficiency. The *subsort* approach can be used to mitigate this. However, this approach becomes infeasible when tuples need to be compared fully, which is often the case when sorted data is used in other relational operations. Therefore, vectorised engines need to explore other solutions to overcome this overhead.

3.6 Sorting Rows in an Interpreted Query Execution Engine

In this section, we discuss and evaluate techniques to improve the sorting performance on row-oriented data in interpreted query execution engines. These existing techniques have been proposed in the literature, but not together and not in the context of sorting relational data in an interpreted query engine, to the best of our knowledge.

[61] Michael W. Blasgen et al. (1977), "An Encoding Method for Multifield Sorting and Indexing"

[4] M. M. Astrahan et al. (1976), "System R: Relational Approach to Database Management"

Normalised Keys. *Key normalisation* [61] is an encoding technique that produces a single order-preserving string from a sequence of values, which dates back to System R [4]. The technique is illustrated in Figure 3.7 for the example query in Section 3.2 (page 25), which orders the `customer` table by `c_birth_country DESC` and `c_birth_year ASC`. The first column, `c_birth_country`, is of type `VARCHAR`. The shorter string 'GERMANY' is padded with '0' such that it has the same length as the longer string 'NETHERLANDS' to ensure that the size of each normalised key is the same. The benefit of having the same size for each normalised key is that they can be swapped in place, which avoids indirection and the random access that it causes[7].

[7] It would also be possible to truncate all strings to a fixed length to avoid excessive padding when string lengths are imbalanced. Prefix comparison ties would then need to be resolved separately.

(a)

c_birth_country	c_birth_year
NETHERLANDS	1992
GERMANY	1924

(b)

c_birth_country	c_birth_year
78 69 84 72 69 82 76 65 78 68 83 0	200 7 0 0
71 69 82 77 65 78 89 0	132 7 0 0

(c)

Normalized Key
177 186 171 183 186 173 179 190 177 187 172 255 128 0 7 200
184 186 173 178 190 177 166 255 255 255 255 255 128 0 7 132

Figure 3.7 Key normalisation. The original data in (a) is represented byte-by-byte as (b) in memory on a little-endian machine. The data is encoded as (c) for `c_birth_country DESC` and `c_birth_year ASC`.

After padding, we flip the bits to get a descending order for this column as specified by the query. The second column, `c_birth_year`, is of type `INTEGER`. On a big-endian machine, the most significant byte comes last. To create an order-preserving encoding, we swap the bytes so that the most significant byte comes first. Then, we flip the sign bit such that negative integers appear before positive integers in the ascending sort order. The resulting normalised keys yield the correct order for the example query if we compare them as we would compare binary strings.

String collations (*e.g.* language-specific comparison rules) are handled by evaluating the collation before encoding the string prefix. `NULL` values are handled by prefixing each value with an additional byte, denoting whether or not the value is `NULL`. This byte is flipped depending on whether we are sorting by `NULLS FIRST` / `LAST`.

We cannot generate a `struct` such as `OrderKey` from Listing 3.3 without JIT compilation. Therefore, we have to use `memcpy` to move the keys. The resulting keys can be compared using the efficient `memcmp` function from the standard library rather than a complex comparison function.

The conversion from columnar data to normalised keys does not come without cost. However, this conversion can be done efficiently by converting one 'block' of vectors at a time, which likely fits in the CPU's cache, and one vector at a time, amortising interpretation overhead. The result is a key that can be compared generically and dynamically without any interpretation or function call overhead.

We implement this approach and run the same benchmark as before, and show the results in Figure 3.8. We can compare this to Figure 3.6, as both figures use the same baseline. The dynamic normalised keys approach performs better than the dynamic *tuple-at-a-time* approach. It can match or even outperform the static *tuple-at-a-time* approach, especially with more key columns and for higher correlation factors.

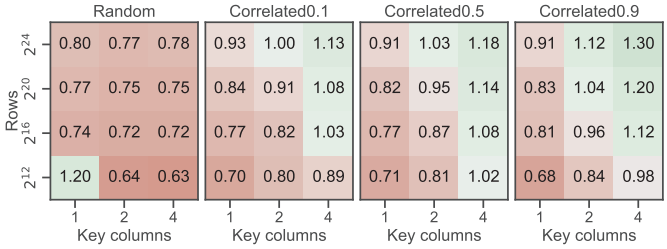

Figure 3.8 Relative runtime (higher is better) of a normalised key approach with a dynamic comparator compared to a static *tuple-at-a-time* comparator on data in row-oriented layout, with `std::sort`.

Radix Sort. Many database systems use quicksort for run generation, as we will see in the next section when we discuss the sorting implementation of the benchmarked systems. Quicksort is a comparison-based sort with a time complexity of $O(n \cdot \log(n))$, where n is the input size. Radix sort, on the other hand, is a distribution-based sort with a time complexity of $O(n \cdot k)$, where k is the key size. This can be faster than quicksort for large inputs, as k does not depend on the input size. The normalised keys illustrated in Figure 3.7 yield the correct sort when sorting with a byte-by-byte radix sort. However, radix sort also has downsides: Radix sort may be slower for a large key size k when n is small. Furthermore, radix sort uses twice as much memory because it needs auxiliary memory of the same size as the input, whereas quicksort sorts in place.

In his survey on implementing sorting in database systems [51], Goetz Graefe remarks that while radix sort may seem like a good option, it has shortcomings that are detrimental for database systems: ① radix sort is most effective if values are uniformly distributed, ② if keys are long and the data contains duplicate keys, many of the passes over the data are unproductive, ③ if input records are nearly sorted, and the keys have a common prefix, the initial passes of radix sort are

[51] Goetz Graefe (2006), "Implementing Sorting in Database Systems"

rather ineffective, and ④ radix sort has worse cache efficiency than comparison-based sorts.

We implement least significant digit (LSD) radix sort and most significant digit (MSD) radix sort that fall back to insertion sort for buckets with ≤ 24 tuples. We add an optimisation to both radix sorts that avoids copying data when all counted values belong to the same bucket, which helps slightly with shortcomings ① and ③. Despite having worse cache performance than MSD radix sort, LSD radix sort is more efficient for smaller keys, and is selected when the key size is ≤ 4 bytes; otherwise, MSD radix sort is selected.

For a fair comparison, we compare our radix sort implementation with the state-of-the-art pdqsort [47], rather than with `std::sort`. The pdqsort algorithm is a highly optimised comparison-based sort that uses optimisations from BlockQuickSort [45] to reduce branch mispredictions, along with recognising worst-case patterns that quicksort traditionally does not perform well on. For both algorithms, we create normalised keys. pdqsort uses `memcmp` dynamically, *i.e.*, with a `size` parameter that is known at runtime, to get a fair estimation of how well these algorithms would perform in an interpreted execution engine. We show the results of this experiment in Figure 3.9.

[47] Orson R. L. Peters (2021), "Pattern-defeating Quicksort"

[45] Stefan Edelkamp et al. (2019), "BlockQuicksort: Avoiding Branch Mispredictions in Quicksort"

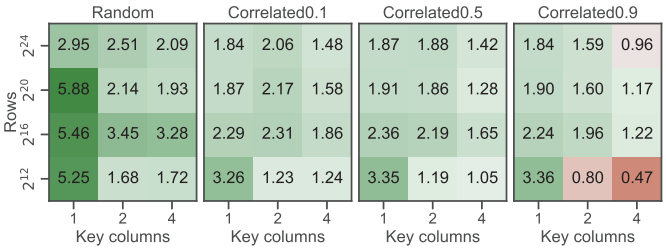

Figure 3.9 Relative runtime (higher is better) of sorting normalised keys with radix sort compared to pdqsort with a dynamic `memcmp` comparator.

For the Random distribution, radix sort performs better than pdqsort on every input and wins out by a considerable margin when there is only one key column. This is no surprise, as radix sort excels at uniform distributions with a high number of unique values, especially when the number of keys is low.

For the Correlated distributions, radix sort is faster than pdqsort for almost all inputs, except some of the highly correlated ones. Radix sort comparatively does not perform well on those inputs, for the reasons mentioned earlier in this section. pdqsort, on the other hand, excels at these inputs because it is able to detect patterns. Despite these characteristics, radix sort is still faster for most inputs, because pdqsort suffers from comparing tuples with a dynamic comparison function, while radix sort does not perform any comparisons.

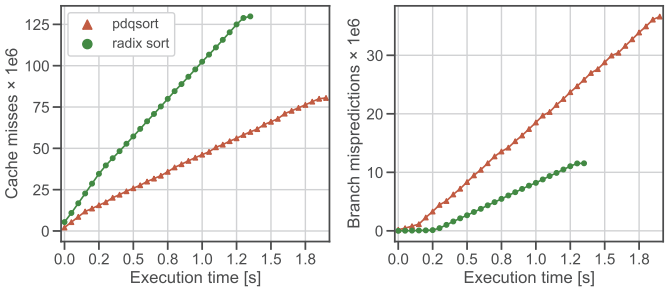

Figure 3.10 Cumulative L1 cache misses and branch mispredictions of sorting 2^{24} rows with 4 key columns, Correlated0.5 distribution, with pdqsort with a dynamic comparison function and radix sort.

We measure performance counters of both sorting approaches for a single input and show them in Figure 3.10. As expected, radix sort has a worse cache performance than pdqsort. The key width is greater than four bytes; therefore, our radix sort implementation chooses MSD radix sort. Radix sort performs better than pdqsort regarding branch mispredictions: it is a mostly branchless algorithm. Branch mispredictions tend to affect performance more than cache misses, as the latency of data cache misses can often be amortised through hardware prefetching when accesses are regular and multiple misses can be overlapped. Branch mispredictions, in contrast, flush the instruction pipeline and must be resolved serially, making their cost difficult to hide.

In conclusion, our micro-benchmarks show that normalised keys and radix sort are promising to address the performance drawbacks of interpreted query engines. In the next section, we investigate if those observations hold up in end-to-end benchmarks on real systems.

3.7 Evaluation

In the previous sections, we applied several existing techniques to sorting relational data and evaluated them in micro-benchmarks. Our results suggest that these techniques can improve the performance of relational sorting in systems with an interpreted query execution engine. Based on these findings, we implemented an efficient relational sort within our analytical database system, DuckDB, which has a vectorised interpreted execution engine. In this section, we first describe DuckDB's sort implementation in detail. Then, we describe the sort implementations of the four other systems under benchmark. Finally, we evaluate and discuss their performance on sorting integers/floats and a relational sort benchmark based on the TPC-DS [62] data generator.

[62] Meikel Poess et al. (2002), "TPC-DS, Taking Decision Support Benchmarking to the next Level"

Implementation. DuckDB's fully parallel sorting pipeline is shown in Figure 3.11. DuckDB uses morsel-driven parallelism [9]; therefore, each thread collects roughly the same amount of data in parallel. Key and payload columns are converted separately to row-oriented layouts, with key columns being converted to normalised keys. This conversion is performed by copying one 'block' of vectors at a time, one vector at a time, making this an efficient and mostly cache-resident process. Both row-oriented layouts use 8-byte alignment, as we have found that this improves the performance of `memcpy`. The rows have a fixed size: variable-sized types like strings are stored separately. Therefore, we encode only a prefix of variable-sized types like strings in our normalised keys: we compare the rest of the string only if the prefixes are equal. For strings, we encode the first n bytes, with n chosen dynamically based on the available statistics on string length, but at most 12. When the amount of data collected by a thread reaches a threshold, we create a sorted run by first sorting the normalised keys with a thread-local radix sort, or pdqsort if there are strings, with `memcmp` as the comparison function. We use a version of pdqsort that is modified to handle our normalised keys, such that we can use inlined tuple comparisons. Then, we reorder the payload, creating fully sorted runs that are added to a thread-global state.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

After all the input data has been added to the global state, we start a 2-way cascaded merge sort. Cascaded merge sort is trivial to parallelise when there are more sorted runs than threads, as we can assign each thread to merge two sorted runs. However, as the runs get merged, parallelisation degrades until a single thread merges the last two sorted runs available. Therefore, we parallelise this phase using the Merge Path [46] algorithm. Merge Path creates partitions that can be merged independently and, therefore, in parallel. The partition boundaries are efficiently computed with a binary search that searches for the 'intersection' of two sorted runs. During merge sort, we compare tuples with `memcmp`.

[46] Oded Green et al. (2014), "Merge Path - A Visually Intuitive Approach to Parallel Merging"

We compare DuckDB's sorting performance with four other OLAP/HTAP engines: ClickHouse [52], MonetDB [53], HyPer [26], and Umbra [54]. OLTP-focused systems like PostgreSQL and MySQL cannot deliver competitive performance in sorting large datasets. Despite their row-major data layouts, their execution engines suffer from large per-value overheads.

[52] Baktagul Imasheva et al. (2019), "The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse"

[53] Stratos Idreos et al. (2012), "MonetDB: Two Decades of Research in Column-oriented Database Architectures"

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

ClickHouse uses a column-oriented layout throughout the sort and performs thread-local sorts with radix sort if sorting by a *single integer column*; otherwise, it uses pdqsort using a tuple-at-a-time comparison approach. JIT compilation is used to reduce interpretation overhead in the comparison function [63]. After the thread-local sorts are done, the sorted runs are merged using a k-way merge. MonetDB also uses a column-oriented layout throughout the sort, using a *single-threaded* quicksort implementation. A subsort approach is used when sorting by multiple key columns. After sorting the key columns, the payload is collected in sorted order. HyPer and Umbra have compiled, row-based sorting implementations similar to what is described in [9]. Threads perform a thread-local quicksort that is similar to pdqsort.

[63] Maksim Kita (2022), *JIT in ClickHouse*

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

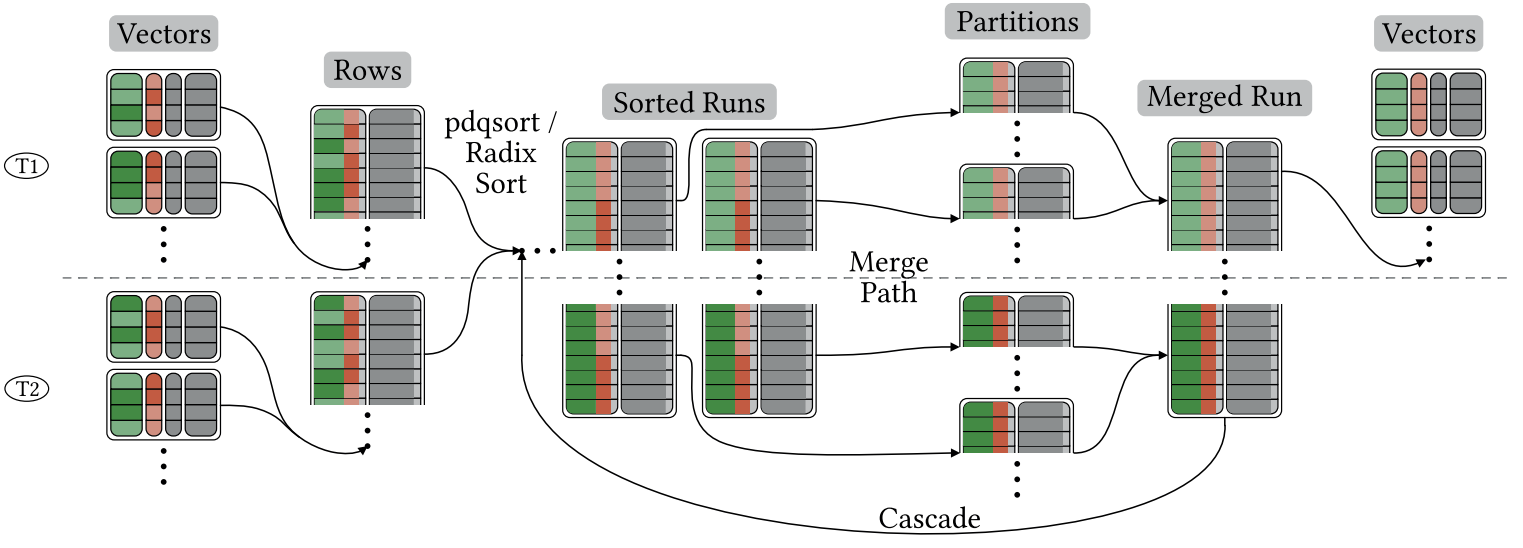

Figure 3.11 Full sorting pipeline in DuckDB. Incoming vectors from worker threads (illustrated as T_1 and T_2 in the figure) are converted to 8-byte aligned row-oriented layouts. Key columns are normalised and stored separately from the payload. The normalised keys are sorted with radix sort or pdqsort, creating sorted runs. The sorted runs are then partitioned and merged in parallel until one run remains. Finally, the result is converted back to vectors.

The results are then merged using a parallel k-way merge. This merge is performed on pointers rather than physically moving the data. The data is physically collected in the sorted order when reading the output of the sort operator.

Benchmarking Relational Sorting. Although sorting can easily dominate the runtime of a query, *isolating* the sorting performance of a database system in a benchmark is difficult because, without access to the source code, we can only reliably observe the end-to-end query runtime. In addition, streaming query execution interleaves the execution of multiple operators, which further complicates isolating sorting performance, even if the system has a query profiler. Measuring end-to-end query runtime introduces unwanted overhead unrelated to sorting, *e.g.*, parsing, optimising, scanning base tables, and transferring the result set through a client protocol. Especially the latter is costly and can easily dominate the query execution time for large result sets [43].

[43] Mark Raasveldt et al. (2017), "Don't Hold My Data Hostage: A Case for Client Protocol Redesign"

Therefore, we want to use a query that produces a small result set and where execution time is dominated by sorting. A full sort is often optimised out of the query plan if it is not strictly needed. For example, `ORDER BY ... LIMIT 1` will typically trigger a specialised top N operator rather than the "normal" sort operator. If we instead aggregate over a subquery that sorts, the optimiser will likely discard the sort because it is irrelevant to the aggregate. We can circumvent this by disabling the optimiser, which is often possible using a configuration setting. This is inadvisable, however, as disabling it may impact performance differently across systems. Instead, we outmaneuver the optimiser with the query shown in Listing 3.4.

Listing 3.4 Query that forces a full sort of the data while only yielding a single output row.

```
SELECT count(payload_column),
FROM (
  SELECT payload_column
  FROM input_table
  ORDER BY
    key_column1,
    key_column2,
    ...,
    key_columnC
  OFFSET 1
);
```

In this query, the count aggregate reduces the size of the result set to 1, making the result set serialisation negligible. The count aggregate reads the sorted subquery, forcing systems that lazily collect a sorted payload to collect it fully. This is an important detail that ensures that all systems under benchmark perform the same amount of work. The count aggregate is cheap to compute and therefore does not add noticeable overhead to the query. Finally, we add the `OFFSET 1`, because we found that the optimisers in the systems we benchmarked do not optimise away the `ORDER BY` in the subquery[8]. We repeat this query 5 times and report only the median end-to-end runtime.

[8] The source code for our end-to-end benchmarks can be found at https://github.com/lnkuiper/experiments/tree/master/sorting, archived at https://doi.org/10.5281/zenodo.15967009. Thanks to André Kohn for suggesting the `OFFSET 1` trick.

Random Integers & Floats. For our first benchmark, we measure raw, single-key sorting performance. We generate two sets of 10 tables containing 10 to 100 million rows in increments of 10 million. The first set contains 32-bit integers from 0 to 99,999,999, shuffled. The second set contains 32-bit floating point numbers between $-1e9$ and $1e9$, sampled from a random uniform distribution. Comparing floats is more expensive than comparing integers; therefore, we expect integer sorting to be slightly faster. Following the findings in our micro-benchmarks, we also expect the performance of the columnar approaches of ClickHouse and MonetDB to degrade more quickly

with the number of tuples than the row-based approaches due to a worse cache performance. We benchmark the sorting performance of all systems using the `m5d.8xlarge` AWS EC2 instance.

We show the results of this benchmark in Figure 3.12. MonetDB is much slower than the other systems; therefore, we cut off the figure to make the differences between the other systems more clearly visible. MonetDB's single-threaded sort takes approx. 29s and 36s for 100 million integers and floats, respectively. For reference, when limited to a single thread, DuckDB takes approx. 10s for the integers and 9s for the floats. As expected, all systems sort floats slightly slower than integers due to the more expensive comparison function of floats, except DuckDB. DuckDB does not suffer from this because it encodes both floats and integers as normalised keys and sorts them the same. DuckDB sorts floats efficiently in this benchmark because the floats have a more uniform distribution, making radix sort more effective.

ClickHouse's sort is competitive with DuckDB, HyPer, and Umbra at 10 million integers, but as the data size increases, its performance degrades more quickly than the other three systems. This is in line with our expectations, as the column-oriented layout that ClickHouse uses causes worse cache performance. DuckDB, HyPer, and Umbra, all of which use a more cache-efficient row-based approach to sorting, show strong scaling with the number of tuples. DuckDB sorts 100 million integers in 1.55s. HyPer and Umbra, which have similar implementations, demonstrate similar performance on this benchmark, with Umbra being slightly faster than HyPer.

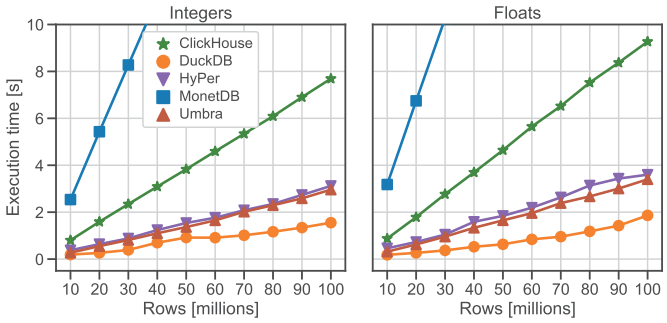

Figure 3.12 Execution times (lower is better) of sorting 10 to 100 million random integers and floats in increments of 10 million.

Table 3.4 Cardinality of TPC-DS tables.

SF	catalog_sales	customer
10	14,401,261	-
100	143,997,065	2,000,000
300	-	5,000,000

TPC-DS Catalog Sales Table. For our second benchmark, we measure how well the systems deal with multiple key columns, which should increase the cost of comparing tuples. Our micro-benchmarks would suggest that the performance of columnar sorting approaches degrades more with additional key columns because they cause more random access than row-based approaches. We use the largest table from TPC-DS, `catalog_sales`, at scale factors 10 and 100. The cardinality of this table can be found in Table 3.4. We select only `cs_item_sk`, and sort it by up to four key columns: `cs_warehouse_sk`, `cs_ship_mode_sk`, `cs_promo_sk`, and `cs_quantity`.

We show the results of sorting the `catalog_sales` table in Figure 3.13. Again, MonetDB's performance is much slower than that of the other systems; therefore, we cut off the figure. MonetDB is roughly 3× slower when sorting by four key columns than by one key column. ClickHouse has a competitive sorting performance when sorting by one key column because it only has random access in one column and uses radix sort. However, when sorting by two key columns, ClickHouse slows down by approximately 4× compared to sorting by one column because it switches from radix sort to pdqsort with a tuple-at-a-time comparison function that has branches and causes random access in both key columns. As expected, DuckDB's, HyPer's, and Umbra's row-based sorting approaches lose less performance here, as they have good cache performance when comparing subsequent key columns. The cost of the comparison function, however, does still matter for the row-based approaches: Umbra is respectively almost 2.5× and 3× slower when sorting four key columns than when sorting one key column at scale factors 10 and 100, while DuckDB and HyPer are only approx. 1.5× slower.

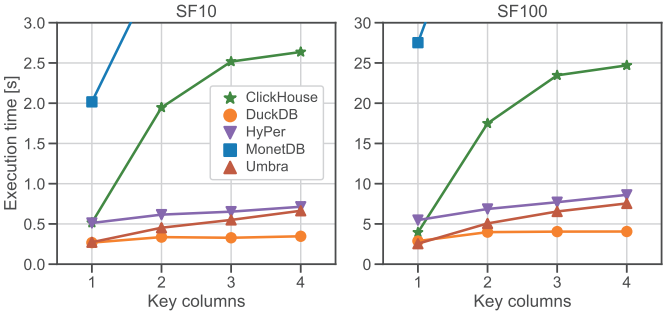

Figure 3.13 Execution times (lower is better) at scale factor 10 and 100 of sorting 1 to 4 key columns (`cs_warehouse_sk`, `cs_ship_mode_sk`, `cs_promo_sk`, `cs_quantity`) of the `catalog_sales` table.

TPC-DS Customer Table. With our third benchmark, we measure how well the systems sort by fixed- vs. variable-sized types. Comparing strings is much more costly than comparing integers; therefore, we expect that sorting strings will be slower. We use the `customer` table from TPC-DS at scale factors 100 and 300. The cardinality of this table can be found in Table 3.4. We select `c_customer_sk` from this table,

and sort it by either `c_birth_year`, `c_birth_month`, and `c_birth_day`, all three of which are `INTEGER` columns, or by `c_last_name` and `c_first_name`, both of which are `VARCHAR` columns.

We show the results of sorting the `customer` table in Figure 3.14. Again, we have cut off the figure so the differences between the systems are more visible. On the x-axis, we have the two categories: integer and string. As expected, sorting strings is slower than sorting integers for all systems. This difference is approx. 3× for ClickHouse and MonetDB, and it is much smaller for the other systems.

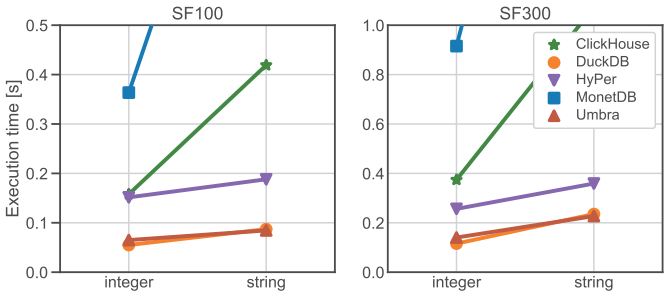

Figure 3.14 Execution times (lower is better) at scale factor 100 and 300 of sorting the `c_birth_year`, `c_birth_month`, and `c_birth_day` columns (integer), and the `c_last_name` and `c_first_name` columns (string) of the `customer` table.

3.8 Payload Handling

The payload columns, *i.e.*, the projected columns that are not part of the `ORDER BY` clause, must be reordered according to the sort order given by the key columns. So far, in this chapter, we have ignored these, as we have focused on the performance of sorting key columns. However, most queries containing an `ORDER BY` clause have many more payload columns than key columns. When the size of these columns exceeds the memory limit, data must be spilled to storage to complete the sort. The initial implementation of a spillable, row-oriented data representation is integrated into DuckDB's sort. This implementation serves as the initial version for an in-memory data representation that can be spilled to storage without large overheads. We have gained valuable insights from working with this initial version, and have drastically improved the data representation since, as will be described in Chapter 4. In this section, we explain the data representation.

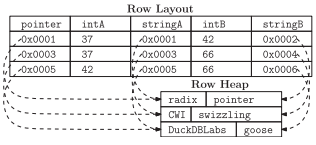

Figure 3.15 DuckDB's fixed-size row layout for payload columns, and accompanying "row heap" containing variable-size data such as strings. The fixed-size rows contain pointers to the variable-size data in the heap.

Initial Layout. DuckDB's initial row layout, shown in Figure 3.15, is optimised for in-memory performance; therefore, rows have a fixed size, such that they can be easily reordered while sorting. With minor modifications, the layout can be spilled to and loaded from disk without full (de-)serialisation. We represent variable-sized columns with pointers into a string heap where the data resides. Each fixed-size row has an additional `pointer` field that points to its accompanying "heap row". This pointer is only used for external sorting.

Row Layout

offset	intA	stringA	intB	stringB
0	37	0	42	5
12	37	0	66	3
24	42	0	66	10

Row Heap

radix	pointer
CWI	swizzling
DuckDBLabs	goose

Figure 3.16 DuckDB's fixed-size row layout for payload columns, *swizzled*. When sorting larger-than-memory data, the pointers are converted to offsets.

To sort more data than fits in memory, we write blocks of already sorted data to disk. Rather than actively doing this in our sorting implementation, DuckDB's buffer manager decides when to do this: when memory is full. Writing data to disk is trivial for fixed-size types, but non-trivial for variable-size types, as the pointers in our row layout will be invalidated when the variable-size data is inevitably loaded back into a different memory address. To be able to offload variable-sized types, we use *pointer swizzling*. When we sort larger-than-memory data, we convert the pointers in the row layout to *offsets*, shown in Figure 3.16.

During external sorting, the 8-byte pointer field is replaced with an 8-byte offset, which indicates where the strings of this row reside in the accompanying row heap. We also replace the pointers to the string values within the row with an 8-byte *relative* offset. This offset denotes how far this particular string is located from the start of this row's heap row. Using relative offsets within rows rather than absolute offsets is useful during sorting: these relative offsets stay constant and do not need to be updated when rows are copied.

With this dual-purpose row-oriented representation, we achieve an almost seamless transition between in-memory and external sorting. The only difference between the two is (un-)swizzling each pointer once, when (un-)pinning the data, which modifies the data in-place and reorders the heap.

External Sorting Evaluation. We will now evaluate the capabilities of DuckDB's external sort. We sort the TPC-DS `catalog_sales` table on `cs_quantity` and `cs_item_sk` at SF 10 and SF 100. We select an increasing number of payload columns to increase the required space. We compare against ClickHouse [52], HyPer [26], Pandas [64] and SQLite [65]. We take the median execution time of 5 queries on a 2020 MacBook Pro with 16 GB of RAM. This experiment tests both the system's ability to sort and reorder the payload. We show the results of this experiment in Figure 3.17.

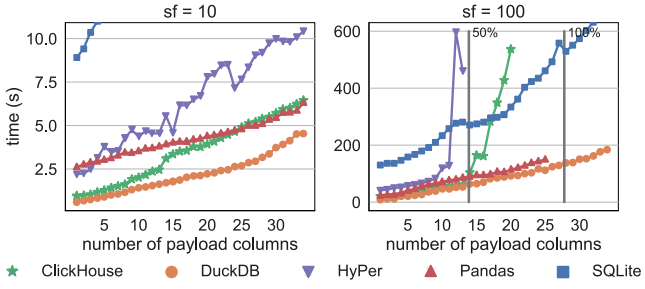

Figure 3.17 `catalog_sales` ordered by `cs_quantity, cs_item_sk`, with an increasing number of payload columns. These columns have few unique values, creating a difficult challenge for sorting algorithms. The grey vertical lines in the SF 100 plot indicate at which points the payload columns take up 50% and 100% of the available memory.

The table fits in memory at SF 10, and the systems' performances are in the same ballpark (except for SQLite), and DuckDB is the clear winner. At SF 100, around 14 selected payload columns, the input table takes up 50% of memory. It is common for systems not to sort

[52] Baktagul Imasheva et al. (2019), "The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse"

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[64] Wes McKinney et al. (2010), "Data structures for statistical computing in Python"

[65] Richard D Hipp (2021), *SQLite*

data in place, but to copy it to a new location, which requires double the amount of memory. The figure shows that all systems except DuckDB and SQLite run into an error due to running out of memory and are unable to complete the benchmark.

ClickHouse and HyPer switch to an external sorting strategy, which are much slower than their in-memory strategies. Therefore, adding payload columns results in a runtime that is orders of magnitude higher. For HyPer, performance degrades sharply when the memory limit is exceeded, after which it improves again slightly, whereas ClickHouse's performance degrades significantly with each added column that goes over the memory limit. Despite switching strategies, both ClickHouse and HyPer run into an out-of-memory error.

Surprisingly, Pandas can load the dataset at SF 100 because macOS dynamically increases swap size. Most operating systems do not do this, and Pandas will not load the dataset on most other OSes. Pandas relies on NumPy's [66] single-threaded quicksort implementation. Pandas shows impressive performance, partly because it already has the input data fully materialised in memory and does not have to stream data through an execution pipeline from the input to the output table like most DBMSes.

[66] Charles R. Harris et al. (2020), "Array programming with NumPy"

Meanwhile, DuckDB and SQLite do not show a visible difference in performance when the data no longer fits in memory. SQLite always opts for a traditional external sorting strategy, resulting in a robust, but overall slower performance than DuckDB.

3.9 Summary

In this chapter, we have discussed the challenges of sorting relational data efficiently in OLAP systems. Following this discussion, we have evaluated different approaches to sorting data using micro-benchmarks. These approaches differed in their row- and column-oriented layouts, their execution engine and tuple comparison method. We have found that sorting data in a row-oriented layout is almost always better than sorting columnar data, mostly due to better cache performance.

Following these findings, we have identified that database systems with a compiled query execution engine can efficiently sort row-oriented data by generating query-specific data types and a comparison function. However, in systems with a vectorised interpreted engine, the efficiency of sorting rows is hindered by interpretation and function call overhead in the comparison function. To overcome this overhead, we have proposed using key normalisation, sorting fixed-width data types with radix sort, and sorting variable-width data types with pdqsort.

Finally, we implemented these techniques in DuckDB. We evaluated our implementation with end-to-end sorting benchmarks. We compared our results on this benchmark with four other analytical database management systems, which have different execution engines. This comparison showed that our sort implementation matches or outperforms all other systems under benchmark and, therefore, that the proposed techniques effectively overcome the interpretation and function call overhead that systems with an interpreted execution engine theoretically would be at a disadvantage of. We also explained how a row-oriented layout could be spilled to disk to enable external sorting, and evaluated it. Our implementation has been part of every DuckDB release since version 0.3.0 and is widely used.

Future Work DuckDB uses pdqsort in its thread-local sorts when strings are present; otherwise, it uses radix sort. Variables other than the data type affect the efficiency of these algorithms, such as key size, number of tuples, the estimated number of unique values, and other statistics. A heuristic that takes these variables into account could improve the algorithm choice. Additionally, pdqsort could be used within the recursive calls to MSD radix sort, which may improve sorting performance even further.

Besides the sort operator, the aggregate, join, and window operators are also blocking operators. They materialise their input because they cannot stream data, *i.e.*, all input data must be consumed and processed before data can be output. In DuckDB, these operators use a unified row-oriented layout. However, DuckDB's vectorised engine moves vectors between operators. When we chain blocking operators, for example, when we aggregate over the output of a join or sort the output of an aggregate, the data is converted from rows to vectors and back to rows again. We could prevent these unnecessary conversions by allowing the execution engine to pass data in row-oriented layout between operators. A similar technique is already being used to reduce the cost of chained joins with many duplicates [67].

[67] Daniel Flachs et al. (2022), "The 3D Hash Join: Building On Non-Unique Join Attributes"

Spilling Intermediates

4.

In the previous chapter, we learned that a row-oriented tuple layout is the most efficient in-memory relational data representation for sorting, even if this requires converting the data from and to a columnar format in systems with a columnar query execution engine. In this chapter, we research how to manage the memory for temporary query intermediates stored using this layout, and how to spill them to storage more efficiently, this time in the context of grouped hash aggregation. With efficient memory management and spilling, performance can degrade more gracefully as the data size exceeds the available memory limit, as illustrated by the plot in Figure 4.1.

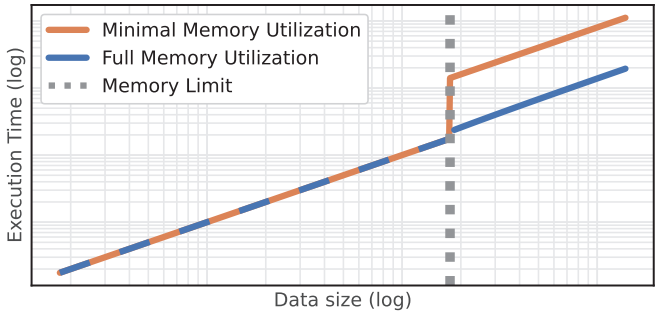

Figure 4.1 Conceptual aggregation performance vs data size (**log-log scale**). When switching from an in-memory strategy to an external strategy that minimises memory usage, performance degradation is harsh and sudden (a "performance cliff"). A unified strategy for in-memory and external aggregation that utilises all available memory degrades more gracefully (performance-robust).

4.1 Introduction

Until late in the 20th century, main memory was much more expensive than secondary storage; therefore, traditional database management systems (DBMS) optimised for disk access, as this was their major bottleneck. "Spillable" data structures like B-trees [68] were not only used to speed up retrieval of persistent data but also inside query operators. As a result, these systems could process workloads that were larger than the small amount of available memory.

[68] Douglas Comer (1979), "Ubiquitous B-Tree"

Around the 1990s, RAM prices decreased, and database systems optimised for main memory [12], for both persistent data and temporary query intermediates [7]. In these systems, main memory access became the bottleneck, and techniques were devised to make better use of CPU caches [11]. DBMS have now evolved into large monolithic database servers, often with large amounts of RAM at their disposal.

[12] H. Garcia-Molina et al. (1992), "Main Memory Database Systems: An Overview"

[7] David J DeWitt et al. (1984), "Implementation Techniques for Main Memory Database Systems"

[11] Peter A. Boncz et al. (1999), "Database Architecture Optimized for the New Bottleneck: Memory Access"

Pure in-memory systems are not economical, however. Efficient utilisation of both memory and secondary storage, *e.g.*, by caching data, is key to providing good performance at a low cost [69]. In recent years, there has been a renewed interest in buffer management, specifically for solid-state memory, which offers higher bandwidth and lower latency than magnetic disk [70, 71, 72]. Data management systems are now reverting to being disk-based without sacrificing in-memory performance [54].

[69] David Lomet (2018), "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed"

[70] Viktor Leis et al. (2018), "LeanStore: In-Memory Data Management beyond Main Memory"

[71] Andrew Crotty et al. (2022), "Are You Sure You Want to Use MMAP in Your Database Management System"

[72] Viktor Leis et al. (2023), "Virtual-Memory Assisted Buffer Management"

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

This body of research has focused on using storage for persistent data but has mostly ignored temporary query intermediates. Analytical (OLAP) systems, which frequently process large volumes of data with large query intermediates, became mainstream after DBMS optimised for main memory. As systems became able to process queries on arbitrary-sized persistent tables, intermediate results could, however, grow to arbitrary sizes depending on the query. In these cases, many modern OLAP systems either abort queries or switch to a traditional disk-based algorithm that is orders of magnitude slower, introducing a "performance cliff", as illustrated in Figure 4.1.

Given the advancements of OLAP systems in the past two decades [18, 26, 9], and the research into buffer management on modern hardware [70, 72], OLAP systems should handle intermediates that exceed main memory more robustly. However, traditional buffer managers have fixed-size pages and a statically allocated pool. This inflexibility makes them undesirable for intermediates; e.g., the bucket array of a large hash table cannot fit on a small fixed-size page. Therefore, temporary data is allocated differently. Managing the entire memory pool, *i.e.*, persistent and temporary data cooperatively, with Cooperative Memory Management [73] (CMM) manner may help systems better utilise available memory.

In this chapter, we go beyond CMM and take a *unified* approach to memory management for persistent and temporary data. We have developed a specialised page layout for temporary data to accommodate such a design. We have integrated the proposed solution into the hash aggregation[1] operator of DuckDB. This enables DuckDB to efficiently utilise secondary storage, enabling larger-than-memory aggregation for inputs with many unique groups.

Contributions. As stated in Chapter 1, the main contributions of this chapter are the following:

- We present a *unified* approach to memory management that permits variable-size pages, and stores pages for temporary query intermediates in the same pool as persistent data, allowing the buffer manager to evict both to storage as needed.
- We present a novel *page layout* for temporary query intermediates on buffer pages, optimised for in-memory performance, and spillable to storage without additional serialisation overhead.
- We integrate the above two techniques into a parallel hash aggregation that can *gracefully degrade performance* as the memory limit is exceeded.

We have implemented the proposed techniques in DuckDB [16]. They have been available since version 0.9.0.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[70] Viktor Leis et al. (2018), "LeanStore: In-Memory Data Management beyond Main Memory"

[72] Viktor Leis et al. (2023), "Virtual-Memory Assisted Buffer Management"

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

[1] We initially integrated this page layout into DuckDB's hash aggregation, but we have since integrated it into the join and sort operators as well.

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

Outline. The rest of the chapter is organised as follows. Section 4.2 introduces the problem of temporary query intermediates and discusses related work. After presenting our approach to memory management that we refer to as *Unified Memory Management* (UMM) in Section 4.3 (page 54), we present our buffer page layout in Section 4.4 (page 56). DuckDB's hash aggregation integrates both of these and is presented in Section 4.5 (page 59). We describe our experimental setup in Section 4.6 (page 65), and take a closer look at how UMM operates in Section 4.7 (page 67). We experimentally evaluate our aggregate implementation and compare it with other implementations in Section 4.8 (page 69). Finally, we summarise the chapter in Section 4.9 (page 74).

4.2 Temporary Query Intermediates

In this section, we discuss the problem of managing temporary query intermediates in the context of related work.

Blocking Operators. Relational operators, like the grouped aggregation operator, are *blocking*. Data cannot be streamed through; therefore, query intermediates have to be *materialised*, *i.e.*, temporarily kept within the operator. As intermediates grow in size, they move down the memory hierarchy. If intermediates grow such that they no longer fit in memory, they are "spilled", *i.e.*, written to storage to complete the query.

Traditional disk-based database systems implemented relational operators for the low-memory environments that were available at the time. Their operators needed to be prepared to spill; therefore, they often used spillable data structures such as B-trees [68]. Storage access degrades query performance as the cost is higher than main memory access. As the available memory grew, operator design was revisited. Hash-based algorithms that operate fully in main memory perform better than disk-based algorithms [7]. Hash tables, unlike B-trees, are not trivially spillable data structures[2].

[68] Douglas Comer (1979), "Ubiquitous B-Tree"

[7] David J DeWitt et al. (1984), "Implementation Techniques for Main Memory Database Systems"

[2] Anything can be spilled using MMAP, but this has several drawbacks, as we will discuss later in this section.

Robustness. Systems that implement hash-based operators may fall back to a disk-based algorithm to complete queries with larger-than-memory intermediates. This choice can be made during query optimisation, *e.g.*, using cardinality estimates, but these are often inaccurate [17]. Alternatively, systems can restart a query when intermediates exceed the memory limit at runtime. Either alternative creates a scenario where adding a *single* row to a table could potentially cause the disk-based algorithm to be chosen. Switching algorithms causes a sudden drop in performance, as the hash-based algorithm operates in memory and has $O(1)$ lookup complexity, while the disk-based algorithm accesses external storage and has $O(\log n)$ lookup complexity if a B-tree is used.

[17] Viktor Leis et al. (2015), "How Good Are Query Optimizers, Really?"

Operator implementations that adapt to larger-than-memory intermediates at runtime provide more robust query runtimes [74]. Hybrid algorithms such as Hybrid Hash Join [22] realise this with a single, efficient algorithm that works regardless of whether intermediates fit in memory. Hybrid algorithms are limited to an input size less than "the square of memory" [7], which is enough for many use cases.

[74] Goetz Graefe et al. (2012), "Robust Query Processing (Dagstuhl Seminar 12321)"

[22] Leonard D. Shapiro (1986), "Join Processing in Database Systems with Large Main Memories"

[7] David J DeWitt et al. (1984), "Implementation Techniques for Main Memory Database Systems"

Memory Management. Traditional database systems implement a buffer manager to move paged data between main memory and external storage. Operators only specify when and for how long pages are needed, and the buffer manager will fetch them from memory or disk. Common wisdom is to use a buffer manager with fixed-size pages and a fixed-size pool, the size of which is user-configured. The fixed size makes them unattractive for temporary query intermediates because large objects, such as hash tables, cannot be stored on a single page. Spreading large objects over multiple pages makes using them less efficient and more complex [54]. Therefore, temporary query intermediates are usually allocated differently [73].

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

This creates two memory pools, one for persistent data on fixed-size pages and one for temporary data that allows variable-size allocations. The fixed-size buffer pool cannot shrink if more memory is needed for intermediates; therefore, the size must be tuned to the workload. CMM [73] challenges the traditional database system buffer manager architecture, but to the best of our knowledge, this has yet to be implemented in a real database system. Allocating from a different pool also means that the buffer manager is not used to offload intermediates, but operators must explicitly read from and write to a temporary file.

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

When query intermediates are spilled from memory to storage, random access time increases. In storage, data is no longer byte-addressable but block-addressable[3]. Going from byte-addressable to block-addressable requires changes to operator algorithm design: blocks must be loaded into memory before the data can be randomly accessed, *e.g.*, by a hash table.

[3] New technologies such as byte-addressable storage or persistent memory may change this in the future, but these are not generally available.

MMAP. One option to circumvent this problem would be to use memory-mapped (MMAP) files, an operating system (OS) feature that maps the contents of a file on secondary storage into a program's address space, making data that resides in storage byte-addressable. MMAP allows the database system to keep using the same algorithm that assumes everything fits in memory by offloading the responsibility of loading and evicting pages to the OS. With this approach, buffers can exceed the available memory limit, which allows, *e.g.*, hash table probes to randomly access data in storage.

MMAP requires the OS, which has no knowledge of the workload, to evict and load pages[4]. Randomly accessing storage is orders of magnitude slower than randomly accessing data in memory and is unlikely to provide robust query performance. Furthermore, the OS's

[4] MMAP is also not easily portable across different OSes, as it requires platform-specific code.

page eviction mechanisms were found not to scale beyond a few threads for larger-than-memory DBMS workloads on high-bandwidth secondary storage devices [71]. This shortcoming also effectively rules out the use of "memory rewiring" [75] to enable larger-than-memory data structures on many-core architectures because it relies on memory-mapped files.

[71] Andrew Crotty et al. (2022), "Are You Sure You Want to Use MMAP in Your Database Management System"

[75] Felix Martin Schuhknecht et al. (2016), "RUMA Has It: Rewired User-Space Memory Access is Possible!"

SSDs. After main-memory database systems became popular in the 1990s, the use of Solid-State Disks (SSDs) became more widely used in the 2010s because of their lower latency and higher throughput. The potential performance benefit of SSDs was quickly recognised in the database literature [13]. Advancements in hardware, as well as advancements in optimising for SSDs, have ultimately led to modern SSD-optimised storage managers such as LeanStore [70]. However, these advancements have focused on using SSDs mainly for persistent storage, not temporary query intermediates. This is partly due to many data management systems having evolved into large monolithic servers running in high-memory environments.

[13] Sang-Won Lee et al. (2008), "A Case for Flash Memory SSD in Enterprise Database Applications"

[70] Viktor Leis et al. (2018), "LeanStore: In-Memory Data Management beyond Main Memory"

External Processing for OLAP. There is a clear need for analytical data management in more economical environments [16]. Many of the developments discussed in this section took place before OLAP became popular in the late 1990s and early 2000s, after main memory database systems had become more widespread. As a result, OLAP systems do not have a long history of processing larger-than-memory intermediates like transactional systems do. Most workloads fit in main memory, but the user experience can often be frustrating if they do not. When the OLAP system runs out of memory to complete a query, it either aborts or switches to a much slower traditional disk-based algorithm. The former leaves the user unable to complete the query, and the latter results in unpredictable query runtimes.

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

Rather than an all-or-nothing approach to memory usage, we argue that analytical database systems should be able to efficiently use *all* available resources to complete a query to provide robust query runtimes and graceful performance degradation if the memory limit is exceeded. We should prioritise the top of the memory hierarchy: use memory when possible and only use storage if necessary. To achieve this, we propose using a novel memory management technique.

4.3 Unified Memory Management

Unified Memory Management, or UMM, is a buffer manager for both persistent and temporary data that permits variable-size allocations. This section first discusses the traditional role of memory management, namely for managing persistent data, and then discusses the novel design that also manages temporary data.

Persistent Data. DuckDB does not allocate a fixed-size buffer pool for persistent data for two reasons:

① As identified in the previous section, a fixed-size pool occupies memory that could potentially be used for temporary query intermediates.

② DuckDB is not a server but an in-process DBMS. Its allocations live within the same address space as the host process. To avoid interfering with the host process, it is important that DuckDB's resource consumption is low when idle; therefore, memory must be deallocated when possible.

For the reasons mentioned in the previous section, DuckDB does not use virtual memory through memory-mapped files. Each buffer is allocated individually.

To avoid fragmentation, DuckDB uses a fixed page size of $2^{18} = 262{,}144$ bytes (256 KiB): all pages for persistent data are of this size. The relatively high page size is chosen for OLAP workloads, which is DuckDB's main use case. 256 KiB is 64 times larger than the default page size of 4 KiB used in CMM and most OLTP systems. If a page is no longer needed, it is added to the eviction queue, which is a lock-free concurrent priority queue with a least-recently-used (LRU) policy. Buffers in this queue are evicted when newly requested memory would cause the memory limit to be exceeded. If enough memory is available, persistent data stays cached in memory. If the newly requested allocation has the same size as a page in the queue that can be evicted, that buffer is reused.

Temporary Data. Allocations for temporary data are more flexible. DuckDB's buffer manager distinguishes three types of temporary allocations: ① Non-paged allocations, ② Paged fixed-size allocations, and ③ Paged variable-size allocations. *Non-paged allocations* are non-spillable allocations of any size. Despite being non-paged, the allocation goes through the buffer manager so that it may decide to evict other pages if the allocation would cause the memory limit to be exceeded, implementing what was proposed as CMM [73]. If the allocation is no longer needed, it is deallocated. *Paged fixed-size allocations* have the same size as persistent pages (256 KiB) and can be efficiently swapped in and out of a temporary file in storage if needed. The temporary file is completely separate from the database file that stores persistent data and could potentially reside on a different storage device if desired. Having the same size for temporary and

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

persistent pages has the advantage of allowing buffers to be reused. *Paged variable-size allocations* can also be written to storage, but because it is of variable size, each is written to a separate temporary file.

Non-paged allocations and paged variable-size allocations are used sparingly only if efficient query processing requires it, *e.g.*, for the buckets of a hash table or strings larger than 256 KiB. Paged fixed-size allocations are the most common and are used to store almost all temporary query intermediates, even if ample memory is available. In our operator implementations, we eagerly destroy temporary pages as soon as they are no longer needed: this deallocates the memory if the page was loaded or frees up disk space if the page was spilled. This prevents the total space used for intermediates (memory + disk) from exceeding the required space.

Because allocation performance can have a large effect on query performance [76], we are cautious about how the design of the buffer manager affects allocation performance. We perform a micro-benchmark in Section 4.7 (page 67).

[76] Dominik Durner et al. (2019), "On the Impact of Memory Allocation on High-Performance Query Processing"

Cooperative Memory Management (CMM). The proposed buffer manager allows all available memory to be used for persistent and temporary data, rather than reserving and managing these separately, like CMM [73]. The key difference, however, is that, unlike UMM, CMM does not have *paged temporary memory*. Because UMM has paged allocations for temporary data, loading persistent data, as well as more allocations for temporary data, can cause not just persistent data but also temporary data to be evicted. Eviction only occurs when memory is full. Another difference is that allocated pages for persistent data are reused for temporary data and vice versa.

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

Compatibility. The proposed buffer manager has a simple API with methods for (un-)pinning pages, similar to other systems. However, because DuckDB uses lightweight compression [77] to compress its columnar storage, it is not generally possible to perform in-place updates, as pages are always fully rewritten. Therefore, the buffer does not implement the notion of *dirty* pages.

[77] Marcin Zukowski et al. (2006), "Super-Scalar RAM-CPU Cache Compression"

4.4 Page Layout

Before giving an overview of the implementation, we first discuss the considerations that went into designing DuckDB's page layout for temporary data. Even in column-oriented systems, a row-oriented layout was shown to be more efficient for join and aggregate hash tables than a column-oriented layout [19]. We showed that it is more efficient for sorting in Chapter 3. Our goal is to achieve good in-memory performance, as most workloads fit in main memory; therefore, we choose to use fixed-size rows in our page layout.

[19] Marcin Zukowski et al. (2008), "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing"

Variable-Size Data. Using fixed-size rows, however, means that variable-sized data types such as strings, which are omnipresent in real-world data [78], cannot be placed within the rows. Variable-sized types are stored elsewhere and can be addressed implicitly, *e.g.*, with an offset, or explicitly, with a pointer stored within the fixed-size row.

[78] Adrian Vogelsgesang et al. (2018), "Get Real: How Benchmarks Fail to Represent the Real World"

Implicit addressing requires additional information, *e.g.*, the ID of the page where the variable-sized data is stored, to locate the relevant data. The use of additional information causes a high degree of indirection and is, therefore, less efficient than explicit addressing. Furthermore, implicit addressing complicates arbitrarily jumping between rows stored on different pages, *e.g.*, for a bucket-chained hash table, as the page ID of the variable-sized data may differ between rows. Therefore, we will only consider explicit addressing to not compromise in-memory performance.

The variable-sized data is stored outside of the row. If it were stored in a global pool[5] that resides in memory, the pool itself could grow to exceed the available memory limit, causing the query to be aborted. MMAP would allow such a global pool to be spilled to storage, but we do not use it for the reasons mentioned earlier. Variable-sized data should, therefore, also be stored on pages to enable spilling. It can be stored either on the same page or on a different page. If stored on the same page, the fixed-size rows could grow from the top, while the variable-sized data grows from the bottom. This can lead to inefficient use of pages, however: If a page has room for more fixed-size rows, but these rows have long strings that do not fit, we move on to the next page, leaving the space on the previous page unoccupied. Therefore, we store fixed-size rows and variable-size data on different pages.

[5] Often called a *heap*.

(De-)Serialisation. Explicit addressing for variable-sized data in combination with storing variable-sized data on buffer pool pages creates a problem of invalid references: if pages storing variable-sized data are evicted and loaded back into memory, their addresses may change, invalidating the explicit addresses in the fixed-size rows pointing to this data. A common way to address such issues is to serialise the data when it is written to storage and deserialise when it is read back into RAM. However, (de-)serialisation can easily

dominate query execution time if not implemented efficiently [43]. Given the current SSD speeds, any (de-)serialisation method may cause spilling to become CPU-bound rather than I/O-bound.

Arrow Flight [79], an efficient data (de-)serialisation format, addresses this problem by only requiring serialisation on write and little to no deserialisation on read. However, Arrow Flight stores data in a column-oriented layout; therefore, using it as our layout would compromise the in-memory performance of our blocking operators [19]. A format like Arrow Flight does not yet exist for row-oriented data. Arrow Flight also specifies a header for each batch, which we do not need as the columns and types of query intermediates are already known.

Serialising before writing, besides being potentially costly itself, also requires the operator to explicitly serialise pages every time they are unpinned, as the buffer manager may evict them at any time. This may result in unnecessary (de-)serialisation costs if the page is not offloaded after all.

Alternatively, if the buffer manager is fully aware of the content stored on the pages, it can serialise the page upon write before evicting it. This has the benefit of serialising the data only when absolutely necessary: only when the data is actually written to storage. However, this requires the buffer manager to know about the content of pages, which will complicate its design and implementation.

Requirements. The previous discussion has argued that using current page layouts for query intermediates will compromise in-memory performance. This is undesirable, as the majority of workloads fit in memory. Using a different page layout for pages in memory and pages in storage will lead to ungraceful performance degradation, as (de-)serialising data is costly. This establishes the need for a page layout that is *both* efficient in memory *and* can be spilled to storage.

From this discussion, we have determined the requirements for the page layout. The page layout must:

1. Use a row-oriented tuple layout with fixed-size rows;
2. Store variable-size data on separate pages;
3. Use explicit addressing for variable-size data;
4. Be spillable to storage without additional serialisation.

In the remainder of this section, we present the design of DuckDB's page layout for temporary data, which meets these requirements. Each tuple is represented by a fixed-size row, the types of which are known when the query plan is generated[6]. This information is stored once, globally, rather than once per page. Fixed- and variable-size data are stored on separate pages. This fulfills requirements 1 and 2. In the remainder of this section, we explain how requirements 3 and 4 are fulfilled.

[43] Mark Raasveldt et al. (2017), "Don't Hold My Data Hostage: A Case for Client Protocol Redesign"

[79] Wes McKinney (2019), *Introducing Apache Arrow Flight: A Framework for Fast Data Transport*

[19] Marcin Zukowski et al. (2008), "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing"

[6] From this, it follows that the widths and offsets of each attribute in the row are also known at this point.

Variable-Size Row. DuckDB uses the optimised 16-byte string representation used by HyPer [26] and Umbra [54]. The first 4 bytes store the length of the string. Small strings of 12 characters or less are *inlined* in the remaining bytes. For *non-inlined* strings (>12 characters), a 4-byte prefix and a pointer are stored[7]. As discussed, a row-oriented layout colocates a tuple's attributes in memory, improving cache efficiency when accessing them subsequently. However, for non-inlined strings, we also have to follow the pointer. To improve locality of reference for these cases, we also store subsequent non-inlined strings in a row sequentially. In other words, each fixed-size row has a corresponding variable-size row. Therefore, the format is optimised for sequential access, even if strings are longer than 12 bytes.

Pointer Recomputation. As discussed, if a page storing variable-size data is spilled to disk and it returns into memory for further processing, it would almost always be loaded at a different location in memory. However, the pointers in the fixed-size rows still point to the previous location. These can be *recomputed* if the previous base pointer of the page is stored. For example, consider a string stored at address `0x48` in a page with base pointer `0x42`, *i.e.*, the string is stored at offset `6` in the page. The page that stores the string is then spilled and loaded back into memory, and now the page has a base pointer of `0x500`. We can recompute the new pointer by subtracting the previous base pointer from the stored pointer and adding the new base pointer: `0x48 - 0x42 + 0x500 = 0x506`. Pointer recomputation can be done in place and *lazily*, *i.e.*, only after we have detected that the variable-size data page has actually gone to disk, instead of carrying out this work preemptively.

Recomputing the pointers within a fixed-size row requires knowing the previous base pointer of the page that stores its corresponding variable-size row. However, storing that pointer within each row introduces an 8-byte overhead. Instead, we can exploit the fact that many subsequent fixed-size rows will have their corresponding variable-size row stored on the same variable-size data page. We could even go as far as allocating one variable-size data page per fixed-size data page, but this requires allocating a non-standard page size to store the variable-size data, which we would like to prevent. If possible, we prefer to allocate the standard page size.

Using the standard page size means that the fixed- and variable-size row pages will not line up. This misalignment is illustrated on the right-hand side of Figure 4.2. Here, many subsequent fixed-size rows have their corresponding variable-size row stored on the same variable-size data page; therefore, storing a pointer in each row would be wasteful. Instead, we store a small amount of in-memory metadata[8], as shown on the left-hand side of Figure 4.2, that describes how the fixed- and variable-size rows on the different pages line up. This metadata also stores the base pointers to variable-size pages, which allows us to recompute the pointers without the overhead of storing a pointer in each row. The explicit addresses are recomputed only if the pages have been spilled to disk, which

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

[7] This string representation is sometimes referred to as the "German String". We show the representation in Listing 4.1.

Listing 4.1 C++ code for the string layout used by DuckDB.

```
union {
  struct {
    uint32_t length;
    char prefix[4];
    char *ptr;
  } pointer;
  struct {
    uint32_t length;
    char inlined[12];
  } inlined;
} value;
```

[8] In our implementation in DuckDB, the data in the pages is only accessed through the metadata, i.e., a pointer to the tuple can only be obtained by calling methods on the class that holds the metadata. This guarantees that the pointers in the tuples are valid when the tuples are accessed. The pointers remain valid as long as the pages are pinned, which is the responsibility of the caller.

we can detect by comparing the stored pointer to the current page pointer; therefore, the performance in RAM is unaffected. In essence, we use implicit addressing, but only upon recomputing the pointers; all further processing uses explicit addressing.

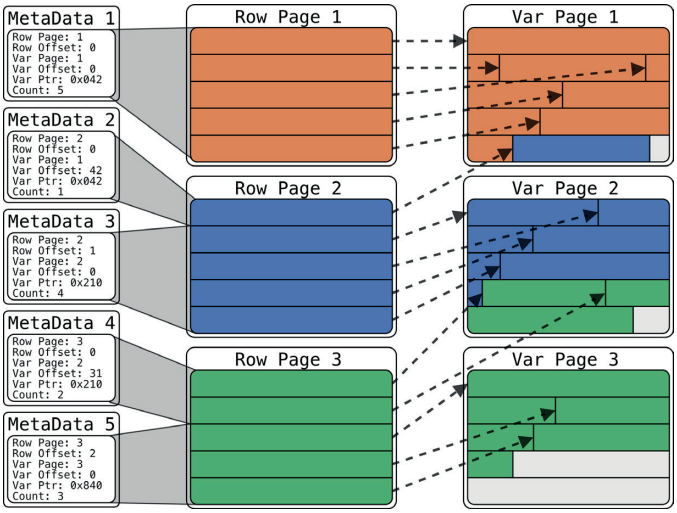

Figure 4.2 DuckDB's page layout for fixed-size rows and corresponding variable-size rows. By using a fixed page size, the row and var pages will not have a one-to-one relation with each other, and a small amount of metadata is needed to describe how they line up. The metadata always describes tuples that have their data stored on the same row and var pages. If a row or a var page is full, another metadata segment is created. This ensures that all variable-size data within a metadata segment has the same base pointer, allowing pointer recomputation to be done per segment.

Any system that uses explicit paged I/O can implement the proposed page layout, as it is only a layer of metadata on top of pages. It fulfills all of the identified requirements, as we use fixed-size rows, store variable-size data on separate pages, use explicit addressing for variable-size data like strings, and can spill the pages to disk without serialisation. The following section explains how our buffer manager and this page layout are integrated into DuckDB's hash aggregation to enable robust external aggregation. This new layout is an improvement over the page layout proposed in Section 3.8 (page 45), as the pointers are recomputed lazily, instead of being swizzled preemptively.

4.5 Robust External Hash Aggregation

In this section, we discuss the considerations that went into designing DuckDB's hash aggregation, before presenting the implementation.

Aggregation is a key operator for OLAP workloads; therefore, its performance is stressed in analytical benchmarks such as TPC-H [80]. If the input data is pre-sorted on the group keys, aggregation can be performed in a streaming fashion [31], otherwise, it is a blocking operator. Grouped aggregation can be resolved by sorting the input data, but if the data fits in memory, hash-based algorithms have been found to perform better [12]. Hashing has the theoretical complexity advantage of $O(n)$ rather than $O(n \log n)$. For these reasons, most OLAP systems implement hash aggregation. As we aim not to impair main memory performance, we implement hash aggregation.

[80] Peter Boncz et al. (2013), "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark"

[31] Thanh Do et al. (2023), "Efficient Sorting, Duplicate Removal, Grouping, and Aggregation"

[12] H. Garcia-Molina et al. (1992), "Main Memory Database Systems: An Overview"

Parallelism. Modern hardware has many available CPU cores. Utilising them well is essential for aggregation performance. *Plan-driven* parallelism [8] decides a static thread count when planning a query, and then lets each thread execute fragments of the query plan. Fragments are connected with a parallelism-aware *exchange* operator that re-routes tuples to different plan fragments. Other operators are kept largely unaware of parallelism; therefore, this approach simplifies operator design. However, re-routing tuples causes overhead, and workloads cannot be balanced if data distributions are skewed [9]. *Morsel-driven* parallelism [9] addresses this with a framework for parallelism that schedules fine-grained tasks on small fragments of input data. Tuples need not be re-routed, and workloads are more evenly balanced across threads. However, operators must be parallelism-aware, complicating their design. Due to its large performance benefits, DuckDB implements morsel-driven parallelism.

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

In morsel-driven parallelism, data is processed in pipelines. Within a pipeline, data moves from an input, such as a table scan, through streaming operators, *e.g.*, projections, to a "pipeline breaker", *i.e.*, a blocking operator, such as hash aggregation. Pipelines themselves are parallel: threads work concurrently on the same pipeline. Operators may have a local state per thread and one state shared across all threads. Cross-thread communication (or even cross-socket communication in the case of NUMA) causes overhead; therefore, accessing the shared state should be done sparingly.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

Low Cardinality Aggregation. Many aggregations reduce the input to just a few rows. For example, TPC-H query 1 always returns just four rows, regardless of the scale factor. Low cardinality aggregations are trivial to perform in parallel in morsel-driven parallelism. Thread-local pre-aggregation is performed in parallel, reducing the input to a few rows per thread, or a single row if the aggregation is ungrouped. After all input data has been read, the data from each thread is combined. Even if there are many threads, combining, *e.g.*, four rows from each thread, has a negligible cost compared to the thread-local pre-aggregation, which could have aggregated millions of rows. Therefore, combining can be done by a single thread without impairing performance.

High Cardinality Aggregation. Aggregation is not limited to low cardinality outputs. Large data volumes can have a large number of unique groups in the output. This leads to high cardinality aggregations: checking whether a column is a primary key, if this is not enforced by the data format, eliminating duplicate rows in machine learning data sets, queries with `DISTINCT`, or grouping by unique customer in a large customer base, to name a few.

When the output has many rows, performing parallel aggregation is nontrivial. After thread-local pre-aggregation, threads have collected large amounts of data. In the extreme case where each group in the input occurs exactly once, the data has not been reduced at all, and

potentially millions of rows remain. Compared to low cardinality aggregation, combining all thread-local data now has a considerable cost, and performing it with a single thread impairs performance.

The authors of morsel-driven parallelism have proposed to take an approach similar to IBM's DB2 BLU [81] and perform thread-local pre-aggregation in a small fixed-size hash table. When this hash table is full, the data is partitioned. After all data has been pre-aggregated and partitioned, the second phase begins. Partitions are exchanged between threads and aggregated independently in parallel[9]. After a thread finishes aggregating a partition, its tuples are immediately pushed into the next pipeline before processing any other partitions. By keeping the thread-local hash table small and fixed-size, fewer cache misses are incurred, and costly hash table resizes are prevented.

[81] Vijayshankar Raman et al. (2013), "DB2 with BLU Acceleration: So Much More than Just a Column Store"

[9] We illustrate DuckDB's approach, which is similar, in Figure 4.3 (page 63).

Data Distributions. This aggregation strategy reduces heavy hitters in skewed data distributions and exploits *interesting orderings* found in real-world data [82], such as many of the same group keys appearing in succession. Because the data is partitioned *after* reducing, this approach to parallel aggregation is more robust to skewed distributions than exchange-based parallelism, which partitions data across threads *before* reducing, creating imbalanced workloads.

[82] P. Griffiths Selinger et al. (1979), "Access Path Selection in a Relational Database Management System"

A downside, however, is that groups would be added to thread-local hash tables multiple times, if they appear at intervals that are larger than the fixed-size hash table; *e.g.*, in random uniform distributions with many unique groups. This scenario would cause memory consumption to grow linearly with the input size, rather than with the output size. Resizing the hash table could prevent this, but this would impair the performance of the more common case of non-uniform real-world data, as it results in more cache misses[10]. Plan-driven parallelism does not suffer from this problem, as the exchange operator always routes the same groups to the same threads, but, as explained, plan-driven parallelism does not perform well in case of skew, which is common in real-world data sets [78].

[10] While the increased memory consumption on random uniform distributions is undesirable, queries will still complete when temporary query intermediates are spillable, as we will see in Section 4.8 (page 69).

[78] Adrian Vogelsgesang et al. (2018), "Get Real: How Benchmarks Fail to Represent the Real World"

External Aggregation. High cardinality aggregations may not fit in RAM on limited hardware or with huge datasets; therefore, intermediates must be spilled to storage to complete the aggregation. As mentioned in the previous section, this can lead to a frustrating user experience when using OLAP systems, as these often only perform well if intermediates fit into memory. If intermediates no longer fit in memory, query performance is not robust, and some queries may not finish at all, as we will see in Section 4.8 (page 69).

Implementation. We now present DuckDB's *embarrassingly external* hash aggregation, which integrates the proposed page layout. This aggregation method is inspired by HyPer's [9] and IBM DB2 BLU's aggregation [81], with key performance optimisations. We describe a partitioned aggregation approach that, as we will see, can trivially process larger-than-memory intermediates without explicitly writing data to storage, leaving the responsibility of spilling to the buffer manager, thereby simplifying operator design. Our design is illustrated in Figure 4.3.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[81] Vijayshankar Raman et al. (2013), "DB2 with BLU Acceleration: So Much More than Just a Column Store"

In the first phase, *Thread-Local Pre-Aggregation*, we assign morsels to threads until all input data has been read. There are usually many more morsels than threads. Data is scanned from morsels in batches of up to 2,048 tuples. Threads pre-aggregate in a small, fixed-size ($2^{17} = 131{,}072$) hash table[11]. The hash table itself consists of two parts. (1) An array of 64-bit entries, which contains a pointer in the lower 48 bits, pointing to the corresponding entry. (2) Temporary pages for hash table entries consisting of groups and aggregates as well as the corresponding pages for variable-length values.

[11] The capacity of this hash table has been determined through extensive benchmarking.

Salt. Pointers have a width of 64 bits on 64-bit CPU architectures, but only the lower 48 bits are used, as this allows for up to approx. 281 TB of address space. We use the remaining 16 bits of the pointer to store the upper 16 bits of the hash of the corresponding tuple, which we call *salt*, shown in the *Hash Table* of the *Salted Linear Probing Hash Table* on the right-hand side of Figure 4.3. Indirect access to the hash table, *i.e.*, storing entries that point to tuples rather than storing tuples directly, keeps the area that is randomly accessed small and allows for specific optimisations that would not be possible otherwise, which will be explained in this section.

Collision Resolution. The offset into the array of 64-bit entries is obtained from the lower bits of the hash. Collisions are resolved using linear probing. Before following the pointer to the tuples and comparing group keys, we first compare the salt. We follow the pointer to compare group keys only if the salt is equal. For uniform hash functions, the salt effectively reduces the chance of having to compare group keys of tuples that do not match by a factor of $2^{16} = 65{,}536$. This approach is most effective with linear probing. In a bucket-chained hash table, different hashes can end up in the same bucket and their salts would have to be combined, e.g., with a bitwise OR, which would flip more bits and, therefore, increase the chance of false positives, or be replaced with a 16-bit bloom filter that uses just 4 bits of the hash. Performance degrades as the hash table fills up due to increased collisions, causing more group key comparisons and more random accesses. This optimisation allows most collisions to be resolved much more efficiently.

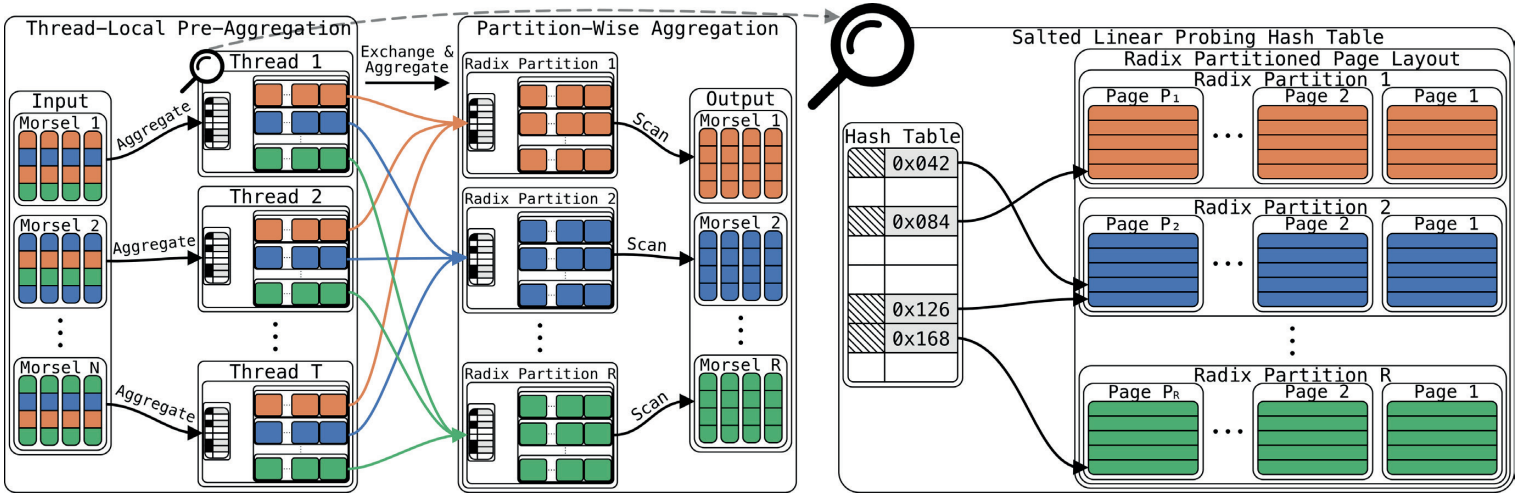

Figure 4.3 DuckDB's hash aggregation. Morsels are assigned to threads until all input data has been read. During phase one, each thread pre-aggregates data in a small fixed-size linear probing hash table with one level of indirection, *i.e.*, offsets obtained from hashes access an array of pointers pointing to tuples. Tuples are radix partitioned and stored using DuckDB's spillable page layout, enabling larger-than-memory aggregation. After pre-aggregation, partitions are exchanged and aggregated partition-wise in parallel. Fully aggregated partitions are immediately scanned, effectively becoming morsels in the next pipeline.

Partitioning. Rather than partitioning tuples when the hash table is reset, like IBM DB2 BLU [81] and HyPer [9] have done, our implementation materialises tuples into partitions that use the proposed page layout. By materialising tuples into partitions, we avoid copying tuples more than once. The partitioned data is stored in row-oriented layout, while the incoming data is stored in column-oriented layout: the conversion of column-oriented to row-oriented takes place simultaneously while partitioning the data. This optimisation is possible because the hash table indirectly accesses tuples as shown in the *Radix Partitioned Page Layout* of the *Salted Linear Probing Hash Table* in the right-hand side of Figure 4.3.

[81] Vijayshankar Raman et al. (2013), "DB2 with BLU Acceleration: So Much More than Just a Column Store"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

The partitions are determined by radix, *i.e.*, a few of the middle bits of the hash that were not yet used for the salt (upper bits) or for the offset into the hash table (lower bits). It is important that the used bits do not overlap, as this would lead to additional collisions and/or reduced effectiveness of the salt. The number of bits used to partition depends on the number of active threads and whether the size of the intermediates is larger than memory, as will be explained later. Because the data is partitioned *after* pre-aggregation, rather than before, partitions are of roughly equal size.

RAM-Oblivious. We reset the hash table once it is two-thirds full. This threshold was empirically determined. Because only the array of 64-bit entries is reset while the tuples stay in place, resetting is an inexpensive operation. The pages that store these tuples can be unpinned[12], as the tuples are no longer active in the hash table. With this approach, partitions are never explicitly written to storage by the aggregation operator, which implementations like IBM DB2 BLU's aggregation [81] would do in low-memory situations. Instead, the buffer manager writes individual pages, rather than entire partitions, to storage when needed. This is especially beneficial under skewed data distributions. We analyse the spilling behaviour of our implementation in-depth in Section 4.7 (page 67).

[12] This allows the buffer manager to evict them when needed.

[81] Vijayshankar Raman et al. (2013), "DB2 with BLU Acceleration: So Much More than Just a Column Store"

Leaving memory management up to the buffer manager simplifies operator design, as the aggregation operator can be largely unaware of the storage location (memory or disk) of the pages it processes. Similar to how cache-oblivious algorithms [83] are oblivious to the size of the CPU caches, our algorithm is practically *RAM-oblivious* during the *Thread-Local Pre-Aggregation* phase: the algorithm's behaviour does not depend on the memory limit. The only requirement is that the small fixed-size hash table fits in memory.

[83] Matteo Frigo et al. (2012), "Cache-Oblivious Algorithms"

During the second phase, *Partition-Wise Aggregation*, the memory limit matters, as the question becomes whether one fully aggregated partition per thread can fit in memory. In practice, this can be achieved by over-partitioning on purpose, *i.e.*, creating many more partitions than threads. This lowers the memory pressure during the second phase. When a partition is fully aggregated, its results are immediately pushed to the next operator, freeing up used pages.

The design of DuckDB's hash aggregation operator should degrade performance gracefully as the size of the temporary query intermediates exceeds the available memory limit, and only the pages that do not fit are spilled to storage. Our aggregate implementation has been part of every DuckDB release since version 0.9.0 and is widely used.

4.6 Experimental Setup

In this section, we describe our experimental setup. All of our experiments are run on AWS EC2 using Ubuntu 22.04.

Hardware. We choose the `c6id.4xlarge` instance because its specifications are similar to affordable hardware, such as a laptop in the early 2020s. This EC2 instance provides an Intel Xeon 8375C CPU with 8 cores (16 threads), 32 GB of DDR4 RAM, and 1 TB of NVMe storage. It is similar to the hardware described in Section 3.3 (page 27), but with half the threads, and a quarter of the RAM. We also use the available Instance Storage, which is physically attached to the host, unlike the more usual Elastic Block Storage on EC2. We set the tenancy of the instance to `dedicated` so that the entire node is reserved, but we do not use the rest of the node's capacity. This setup eliminates the *noisy neighbor* problem that cloud environments may have, improving the consistency of our results.

Data. We create a grouping benchmark using the data generator from TPC-H. We generate the `lineitem` table at scale factors 1, 2, 4, 8, 16, 32, 64, and 128. At these scale factors, the row count ranges from 6,001,204 to 768,046,921, and the size of the generated CSV ranges from 0.72 GB to 96.72 GB.

Query. Although aggregation can dominate the runtime of a query, *isolating* the performance of any relational operator in a benchmark is difficult because we can often only reliably observe end-to-end query runtime. This is even the case for systems with a query profiler, because the execution of multiple operators is interleaved in streaming query engines. Measuring end-to-end query runtime introduces unwanted and unrelated overheads, *e.g.*, small overheads such as parsing and optimising, scanning base tables, and transferring the result set through a client protocol. The latter is especially costly for large result sets, *e.g.*, high cardinality aggregations, and can easily dominate query execution time [43].

[43] Mark Raasveldt et al. (2017), "Don't Hold My Data Hostage: A Case for Client Protocol Redesign"

Instead of transferring the result set through a client protocol, we could write the result to a table instead, *i.e.*, `CREATE TABLE ... AS`. This is slightly better but also introduces a large amount of overhead in the form of bulk insertion, which may create a large overall performance difference across different systems. Adding a cheap, ungrouped aggregate on top of the expensive aggregate that we want to measure would be even better, but in some cases, this allows query

optimisers to remove unused columns, potentially leading to unequal plans across different systems. Therefore, we have devised the benchmark query shown in Listing 4.2.

Listing 4.2 Aggregation benchmark query that forces an aggregation to be fully computed while only yielding a single output row. Note that we precompute `N`, the size of the result, such that `OFFSET N - 1` yields a single row without the need to add `LIMIT 1`.

```
SELECT
  group_key1,
  group_key2,
  ...,
  group_keyG,
  ANY_VALUE(col1),
  ANY_VALUE(col2),
  ...,
  ANY_VALUE(colC)
FROM lineitem
GROUP BY
  group_key1,
  group_key2,
  ...,
  group_keyG
OFFSET N - 1;
```

Here, `N` is the number of unique groups in the query, which has been precomputed for each grouping. This yields a result set of exactly one row. This query is underspecified, and any single row of the input satisfies the query. However, the number of unique groups is not known beforehand; therefore, systems are forced to fully process and discard the first `N - 1` groups before emitting a single row. Additional columns other than group keys are selected using the `ANY_VALUE` aggregate function to increase the memory pressure without changing the number of unique groups.

Table 4.1 Output size of grouping lineitem by different combinations of columns. An output size of 25.00% means that the number of rows in the output is equal to 0.25 times the input size.

Number	Grouping	Size
1	l_returnflag, l_linestatus	4 rows
2	l_partkey	3.33%
3	l_partkey, l_returnflag, l_linestatus	10.58%
4	l_suppkey, l_partkey	13.33%
5	l_orderkey	25.00%
6	l_orderkey, l_returnflag, l_linestatus	34.87%
7	l_suppkey, l_partkey, l_returnflag, l_linestatus	36.17%
8	l_suppkey, l_partkey, l_shipinstruct	45.34%
9	l_suppkey, l_partkey, l_shipmode	61.83%
10	l_suppkey, l_partkey, l_shipinstruct, l_shipmode	88.56%
11	l_orderkey, l_partkey	99.99%
12	l_orderkey, l_suppkey	99.99%
13	l_suppkey, l_partkey, l_orderkey	100.00%

Groupings. We group the `lineitem` table by different combinations of columns, shown in Table 4.1. We have two variants of each grouping. The *thin* variant selects only the group columns. The *wide* variant selects all other columns using the `ANY_VALUE` aggregate function.

With these groupings, the thin and wide variants, and different scale factors, this benchmark represents a wide variety of aggregations, ranging from low to high memory pressure. For example, the lowest memory pressure is the thin variant of grouping by `l_returnflag`, `l_linestatus`, which only materialises four rows of two columns. The highest memory pressure is the wide variant of grouping by `l_suppkey, l_partkey, l_orderkey`, which materialises the entirety of the `lineitem` table. All of our experiments are publicly available on GitHub[13].

[13] Experiments: https://github.com/lnkuiper/experiments/tree/master/oocha, archived at https://doi.org/10.5281/zenodo.15967009.

4.7 Loading, Spilling & Allocating

In this section, we take a closer look at how the proposed memory management operates.

Loading & Spilling. We first examine how DuckDB's buffer manager decides to load and spill pages when running an aggregation workload. The workload we have selected for this experiment is thin grouping 4 at scale factor 128, because it requires materialising a fair amount of intermediate data, but not so much that the entire aggregation becomes bottlenecked by I/O. This grouping only selects `l_orderkey`, which is just over 1.5 GB in DuckDB's file format, which uses lightweight compression techniques [77]. We consider two scenarios: In the first scenario, a single connection runs this grouping 10 times in a row, with DuckDB configured to have 4 threads and a memory limit of 3.5 GB. DuckDB is rarely used as a server: a single connection is a common use case for local data set analysis. We have chosen a memory limit of 3.5 GB, as this is approximately the total size of the intermediates needed for this grouping and, therefore, necessitates evicting pages. This memory limit yields more interesting results than a high or low memory limit, as, in theory, not much I/O is needed to complete the query, but wrong decisions by the buffer manager may, however, cause unnecessary I/O. In the second scenario, four connections simultaneously run this grouping 10 times in a row, with DuckDB configured to use $4 \times 4 = 16$ threads and a memory limit of $4 \times 3.5 = 14$ GB.

[77] Marcin Zukowski et al. (2006), "Super-Scalar RAM-CPU Cache Compression"

As mentioned in Section 4.3 (page 54), DuckDB's *eviction policy* is to evict pages using a concurrent LRU queue. Pages of all types are added to the same queue: persistent, temporary, fixed- and variable-size; no distinction is made between the types of pages. Eviction policies have been researched in depth in the context of traditional buffer pools that only store persistent data, but not in the context of temporary data. These policies try to keep often-used persistent data in memory to speed up future access. Temporary data is short-lived; therefore, having a different policy for these pages may be beneficial. We will refer to DuckDB's default eviction policy as the *Mixed* policy. For this experiment, we implement two more eviction policies, which store persistent and temporary pages in two separate LRU queues. *TemporaryFirst* evicts temporary pages before evicting persistent pages. *PersistentFirst* evicts persistent pages before temporary pages. We show the results of this experiment in Figure 4.4.

With a single active connection, Mixed, TemporaryFirst, and PersistentFirst have similar respective execution times of 69.8s, 72.1s, and 66.8s. The size of the temporary file was observed to stay under 500 MB for PersistentFirst and under 2 GB for TemporaryFirst and Mixed. Persistent data is read at the start of each query during thread-local pre-aggregation. Only temporary data is needed during partition-wise aggregation once all data has materialised. Evicting persistent data does not require writing to storage because it is already stored in

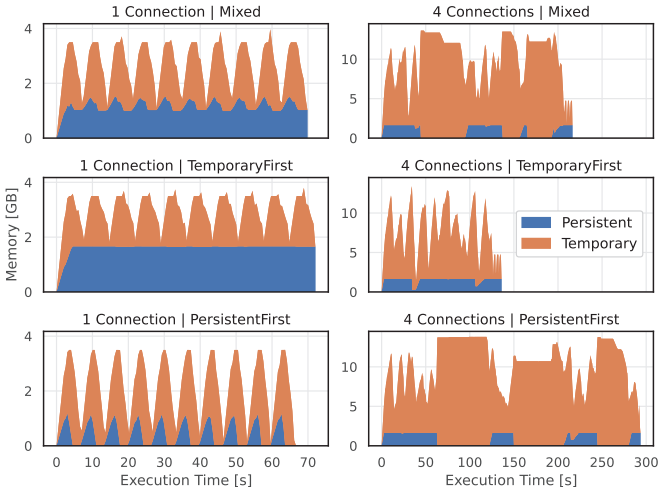

Figure 4.4 Visualisation of DuckDB's memory when repeatedly performing thin grouping 4 at scale factor 128 in a single-connection and multi-connection scenario with different buffer eviction policies.

the database file. Evicting temporary data, on the other hand, requires writing it to storage and is, therefore, more costly. PersistentFirst is therefore the most efficient strategy when a single connection is active.

With four active connections, Mixed, TemporaryFirst, and PersistentFirst have varied respective execution times of 216.4s, 135.9s, and 293.5s. This order is the opposite of the previous scenario. The size of the temporary file was observed to stay under 8 GB for all three policies. In Figure 4.4, it appears that all persistent data is evicted at some point, but some of it is always loaded into memory, although the amount is so small that it is not visible. When most persistent data is evicted, thread-local pre-aggregation slows down because every read requires accessing storage. This behaviour causes memory to clog up with temporary data, and overall query throughput suffers due to excessive thrashing, *i.e.*, loading and spilling pages repeatedly. In the remainder of our experiments, DuckDB uses the Mixed eviction policy as a decent compromise between PersistentFirst and TemporaryFirst.

Allocation Performance. Allocation latencies can significantly affect query performance [76]; therefore, we perform a micro-benchmark to investigate how the proposed buffer manager affects allocations, similar to what was done for CMM [73]. We measure the time it takes to allocate a small region of 262,144 bytes, which is DuckDB's fixed page size, and the time it takes to allocate a large region of 268,435,456 bytes, which is 1,024 times larger. Timings are averaged over 1,024 allocations. Simply allocating this using `jemalloc` [84], DuckDB's internal allocator, takes 1.5 microseconds for the small region and 1.7 microseconds for the large region. When routing these allocations through DuckDB's buffer manager, when ample memory is available, these allocations take 1.7 microseconds and 2.0 microseconds, respectively. The overhead is negligible and can be explained by

[76] Dominik Durner et al. (2019), "On the Impact of Memory Allocation on High-Performance Query Processing"

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

[84] Jason Evans (2006), "A scalable concurrent malloc (3) implementation for FreeBSD"

bookkeeping, such as incrementing the atomic counter for the current memory usage.

When we fill up memory by loading persistent data, performing these allocations will cause pages to be evicted. Performing the small allocation takes less time in this scenario: 0.9 microseconds. Only one persistent page needs to be evicted, which is for free because persistent pages are replicated in storage, and the allocation can immediately be reused.

Performing the large allocation, however, now takes 0.9 *milli*seconds because it causes 1,024 persistent pages to be evicted. Unlike the previous scenario, the pages cannot be reused and must, therefore, be deallocated. Deallocations cause significant overhead, as was also found to be the case for CMM [73]. If a page size of 4 KiB was used rather than DuckDB's default page size of 256 KiB, the large allocation would have caused 65,536 deallocations instead, causing even more overhead.

[73] Robert Lasch et al. (2023), "Cooperative Memory Management for Table and Temporary Data"

Given that the deallocation overhead only occurs when memory is full and only for the variable-size allocations that are sparingly used in DuckDB, we find this overhead acceptable.

4.8 Evaluation

In this section, we experimentally evaluate our implementation and compare it with other implementations. We compare our system, DuckDB [16] (version 0.9.2) against three other systems with strong aggregation performance:

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

ClickHouse [52] (version 23.11.1), an open-source column-oriented DBMS for OLAP workloads.

[52] Baktagul Imasheva et al. (2019), "The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse"

HyPer [26] (version 2023.3), a main-memory-based relational DBMS for mixed OLTP and OLAP workloads, which uses data-centric code generation, developed at Technische Universität München (TUM), now Tableau's data engine.

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

Umbra [54] (v0.1 2023-11-21), HyPer's successor, a disk-based DBMS with in-memory performance, which also uses data-centric code generation, also developed at TUM.

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

We do not compare with database systems such as PostgreSQL or MySQL as these are orders of magnitude slower than modern OLAP systems at aggregation for various reasons, such as large amounts of per-tuple overhead[14] [18].

[14] Despite being an OLAP system, we do not include MonetDB due to poor performance.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

We have verified that the aggregation query described in Section 4.6 (page 65) leads to the same query execution plan in all systems in the benchmark. We run each query five times and report the median execution time. If queries do not finish within 10 minutes, they are timed out, *i.e.*, stopped.

System Grouping \ SF	Du	Cl	Hy	Um	Du	Cl	Hy	Um
	2				8			
1	**0.01**	0.08	**0.01**	0.02	**0.03**	0.27	0.05	0.04
2	0.08	**0.04**	0.13	**0.04**	0.44	**0.16**	0.66	0.19
3	**0.13**	0.30	0.20	**0.13**	0.58	1.21	1.00	**0.51**
4	0.12	0.08	0.16	**0.07**	0.58	0.29	0.83	**0.28**
5	**0.05**	0.08	0.07	**0.05**	0.18	0.29	0.36	**0.16**
6	**0.08**	0.35	0.21	0.15	**0.30**	1.42	0.81	0.55
7	**0.17**	0.37	0.28	0.23	**0.72**	1.56	1.36	0.87
8	0.23	0.36	0.27	**0.20**	0.97	1.51	1.39	**0.78**
9	**0.16**	0.37	0.30	0.21	**0.74**	1.53	1.58	0.78
10	**0.25**	0.50	0.41	0.38	**1.10**	2.06	1.96	1.50
11	**0.13**	**0.13**	0.37	0.18	0.59	**0.51**	1.71	0.65
12	**0.13**	**0.13**	0.36	0.18	0.59	**0.52**	1.73	0.65
13	**0.15**	0.17	0.38	0.22	**0.64**	0.70	1.80	0.79
Norm. Geom. Mean	1.00	1.69	1.80	1.13	1.00	1.53	1.98	0.98
Grouping \ SF	32				128			
1	**0.14**	1.10	**0.14**	0.16	**0.54**	4.06	**0.54**	A
2	2.03	0.75	2.86	**0.68**	10.80	**4.04**	12.83	A
3	2.78	6.35	4.28	**2.24**	**14.49**	42.47	348.10	A
4	2.86	1.63	3.38	**1.22**	22.06	**9.80**	231.86	A
5	0.74	1.57	1.66	**0.54**	**3.17**	9.41	6.86	A
6	**1.27**	7.32	3.27	2.12	**5.80**	48.79	213.41	A
7	**3.42**	8.32	5.61	4.32	**24.62**	51.57	457.52	A
8	5.25	8.11	5.72	**3.44**	65.97	**49.49**	412.86	A
9	3.59	8.01	6.42	**3.47**	**32.77**	46.51	444.50	A
10	**5.78**	11.34	169.63	14.98	89.27	**77.27**	576.68	A
11	2.72	**2.59**	6.94	2.86	21.38	**18.43**	413.54	A
12	2.65	**2.59**	6.80	2.84	21.89	**18.22**	411.58	A
13	**2.87**	3.64	7.24	3.39	**22.20**	28.31	432.33	A
Norm. Geom. Mean	1.00	1.69	2.17	0.94	1.00	1.48	8.74	A

Table 4.2 Execution time in seconds for the *thin* variant of all groupings at scale factors 2, 8, 32, and 128. Lower is better. The lowest execution times are highlighted in bold. 'A' denotes that the query was aborted. We summarise by normalising execution times to DuckDB and then taking the geometric mean.

[85] Philip J. Fleming et al. (1986), "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results"

Thin Groupings. Table 4.2 shows the results for the thin variant of all groupings. We show only scale factors 2, 8, 32, and 128, as a wider table does not fit. We summarise the execution times per scale factor by first normalising to (*i.e.*, dividing by) DuckDB's execution time and then taking the *geometric mean* across queries, as this weighs each query fairly [85].

For scale factors 2, 8, and 32, the data fits entirely in memory, and the execution times of the systems, as well as the normalised geometric means, are similar, with DuckDB and Umbra being the fastest. One exception is HyPer for grouping 10 at scale factor 32, which takes an order of magnitude longer than the other three systems. For this grouping, HyPer has already switched to its external aggregate implementation. We observed HyPer's external aggregate to use a minimal amount of memory, less than 2 GB.

For scale factor 128, some of the larger groupings no longer fit in memory. The exact point where this happens differs per system. For almost all groupings at this scale factor, HyPer uses its external aggregation operator, which is much slower than its in-memory ag-

gregation operator. Therefore, HyPer can barely finish some of the queries within the 600-second timeout, resulting in a much higher normalised geometric mean. Umbra does not yet have an external aggregation implementation and has to abort queries at this scale factor. Both DuckDB and ClickHouse are able to finish all groupings well within the 600-second timeout.

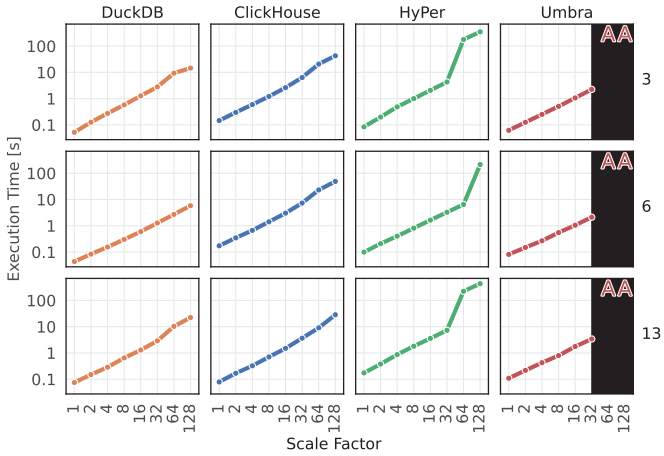

Figure 4.5 Execution times for the *thin* variant of groupings 3, 6, and 13 at scale factors 1 through 128 (log-log scale). Lower is better. 'A' denotes that the query was aborted.

We now look closer at the scaling behaviour for specific thin groupings. In Figure 4.5, we show the execution times for the thin variants of groupings 3, 6, and 13 at scale factors 1 through 128. Note that both the x- and y-axis are on a logarithmic scale. The data fits in memory up to scale factor 32, and all four systems show linear scaling, as expected from hash aggregation. HyPer switches to external aggregation for all three groupings at scale factor 128, and, surprisingly, despite yielding a smaller result size than grouping 6 and 13, HyPer uses external aggregation for grouping 3 at scale factor 64. This results in a significant performance cliff, as execution time increases more than 10×, despite the data size growing by only 2×. For DuckDB and ClickHouse, there is a much less noticeable performance *bump*, as execution times only increase by approx. 3×. Umbra runs out of memory and is unable to complete the queries at scale factors 64 and 128.

These results show that switching from an in-memory to an external algorithm is not robust, as it causes performance cliffs: large and sometimes unpredictable spikes in performance. DuckDB's hash aggregation is much more robust.

System	Du	Cl	Hy	Um	Du	Cl	Hy	Um
Grouping \ SF	2				8			
1	0.04	0.19	0.03	**0.02**	0.16	0.67	0.10	**0.06**
2	0.42	0.34	0.23	**0.22**	2.25	1.45	1.07	**0.77**
3	0.43	0.59	0.27	**0.23**	2.30	2.56	1.25	**0.91**
4	0.53	0.51	0.27	**0.23**	2.77	2.43	1.28	**0.91**
5	0.25	0.58	0.23	**0.13**	0.80	2.96	0.89	**0.44**
6	0.23	0.72	0.29	**0.17**	1.01	3.24	1.22	**0.67**
7	0.51	0.75	0.35	**0.29**	2.56	3.34	1.76	**1.22**
8	0.87	1.01	0.42	**0.36**	3.94	4.57	1.96	**1.47**
9	0.59	0.88	0.47	**0.32**	3.22	4.24	2.18	**1.41**
10	0.75	1.00	0.61	**0.42**	3.69	4.91	2.69	**1.74**
11	0.64	0.83	0.61	**0.38**	3.22	3.75	2.71	**1.60**
12	0.62	0.79	0.62	**0.38**	3.32	3.76	2.70	**1.58**
13	0.66	0.83	0.60	**0.38**	3.20	3.85	2.67	**1.59**
Norm. Geom. Mean	**1.00**	**1.53**	**0.77**	**0.54**	**1.00**	**1.45**	**0.70**	**0.45**
Grouping \ SF	**32**				**128**			
1	0.63	2.57	**0.38**	A	2.57	9.91	**1.52**	A
2	52.79	**18.41**	211.11	A	347.28	**111.66**	499.04	A
3	30.00	**26.39**	243.30	A	256.29	**133.50**	T	A
4	53.46	**26.19**	255.55	A	350.96	**122.97**	T	A
5	4.63	30.16	**3.13**	A	**67.56**	A	287.87	A
6	4.96	33.20	**4.56**	A	**72.69**	150.75	378.23	A
7	**30.61**	31.85	382.80	A	**260.12**	A	T	A
8	68.49	**38.79**	487.08	A	**407.26**	A	T	A
9	46.12	**36.37**	451.81	A	**331.08**	A	T	T
10	64.84	**43.70**	585.54	A	**399.46**	A	T	A
11	60.04	**36.41**	530.67	A	**396.53**	A	T	A
12	60.73	**36.00**	533.54	A	**382.09**	A	T	A
13	54.55	**37.00**	534.56	A	**359.95**	A	T	A
Norm. Geom. Mean	**1.00**	**1.06**	**4.54**	**A**	**1.00**	**A**	**T**	**A**

Table 4.3 Execution time in seconds for the *wide* variant of all groupings at scale factors 2, 8, 32, and 128. Lower is better. The lowest execution times are highlighted in bold. 'A' denotes that the query was aborted. 'T' denotes that the query timed out after 600 seconds. We summarise by normalising execution times to DuckDB and then taking the geometric mean.

Wide Groupings. Table 4.3 shows the results for the wide variant of all groupings at scale factors 2, 8, 32, and 128. Compared to Table 4.2, there is a clear difference, as all execution times are higher due to having to scan and materialise the additionally selected columns. Like before, the execution times of the four systems are similar at the lower scale factors, 2 and 8, with Umbra being the clear winner, as evidenced by the execution times and normalised geometric mean. Out of the four systems here, DuckDB is the only system that does not use just-in-time (JIT) compilation for aggregate functions. This decision has been made because of portability considerations. JIT compilation is known to speed up aggregation performance [27], which could explain why DuckDB's execution times are slightly higher.

At scale factor 32, there are already significant differences in execution time. HyPer switches to its external aggregation operator for most queries due to the increased memory pressure caused by selecting additional columns. As a result, HyPer is slower than ClickHouse and DuckDB by an order of magnitude. Umbra cannot complete any groupings at scale factor 32 or higher.

[27] Timo Kersten et al. (2018), "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask"

ClickHouse performs well at scale factor 32, having the fastest execution time on most queries and almost the lowest normalised geometric mean if it were not for groupings 5 and 6. However, its strategy to achieve this performance does not hold up, as ClickHouse has to abort most queries at scale factor 128. At scale factor 128, DuckDB is the only system that can complete the benchmark within the 600-second time limit and the 32 GB of main memory available.

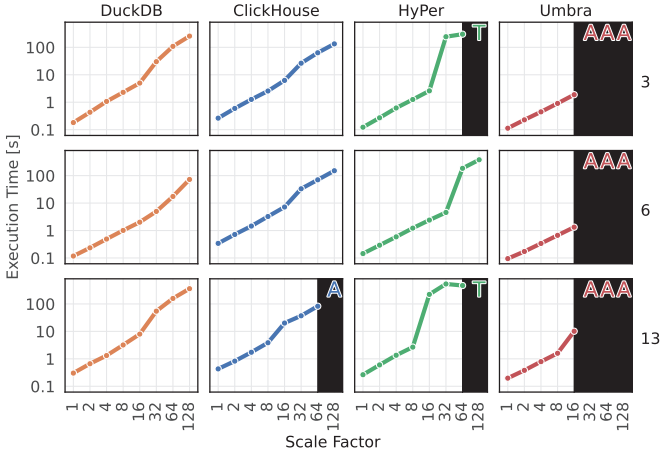

Figure 4.6 Execution times for the *wide* variant of groupings 3, 6, and 13 at scale factors 1 through 128 (log-log scale). Lower is better. 'T' denotes that the query timed out after 600 seconds. 'A' denotes that the query was aborted.

We again look at the scaling behaviour for the same groupings as before, 3, 6, and 13, but now the wide variants, in Figure 4.6. All systems show linear behaviour for the first few scale factors. However, performance degradation starts earlier, at around scale factor 32, although this depends on the number of rows the grouping yields. HyPer switches to external aggregation for the higher scale factors for all three groupings. It switches at scale factors 32, 64, and 16 for groupings 3, 6, and 13, respectively. The external aggregation sharply degrades performance compared to the in-memory aggregation and finally causes HyPer to time out at scale factor 128 on two out of three groupings. Umbra is not able to complete any of the groupings at scale factor 32 and above.

ClickHouse scales well, especially for the higher scale factors, and although its aggregation strategy can utilise storage to be able to process larger-than-memory intermediates, it still runs out of memory for the largest grouping, grouping 13, at scale factor 128[15]. DuckDB does not have this problem, and again, although DuckDB has a slight performance bump around scale factor 32/64, it can complete all groupings at the highest scale factor without problems, with more than three minutes to spare until the 600-second timeout.

[15] Although ClickHouse has an external aggregation implementation, it could not manage to complete some queries, even with different combinations of the `max_memory_usage` and `max_bytes_before_external_group_by` settings.

4.9 Summary

In this chapter, we discussed the impact of large temporary query intermediates, specifically for OLAP systems. We identified that current approaches for intermediates have limitations that impair memory utilisation and efficient spilling. To address these problems, we proposed Unified Memory Management, which unifies memory management for temporary and persistent data and a page layout that can be lazily spilled without serialisation overhead. Together, these techniques allow blocking operators to utilise all available memory for intermediates and to efficiently and lazily spill them, which allows operator implementations to process larger-than-memory intermediates without sacrificing in-memory performance.

We integrated both techniques into DuckDB's parallel hash aggregation. We experimentally evaluated our implementation and compared it against three systems with strong aggregation performance, using a benchmark with a diverse aggregation workload and varying numbers of unique groups. The results show that DuckDB can aggregate larger-than-memory intermediates without falling off a proverbial performance cliff. This achievement can be attributed to the proposed techniques in tandem with the high I/O throughput of modern SSDs. The results also show that DuckDB's aggregation is competitive when intermediates fit in main memory, demonstrating that robust external aggregation performance can indeed be achieved without sacrificing in-memory performance.

Future Work. The proposed techniques were implemented and evaluated for OLAP. More research is needed to determine their viability for OLTP, where smaller page sizes, dirty pages, and high numbers of concurrent writers are common.

Our experimental results showed that the efficiency of eviction policies is workload-dependent. Prior research in this area has not considered evicting temporary data, only persistent data. More research into eviction policies for UMM is needed to determine the most efficient strategies.

As mentioned in Section 4.5 (page 59), the same group may be materialised multiple times during thread-local pre-aggregation, possibly across multiple threads, due to morsel-driven parallelism. These additional materialisations can cause the size of the temporary query intermediates to grow large, potentially causing unnecessary I/O. This can be mitigated by an adaptive strategy that could, for example, exchange and aggregate partitions early during this first phase if the memory limit would otherwise be exceeded, with the objective to reduce the size of the intermediates.

5. Query Plan Memory Management

In the previous chapter, we saw how unifying the memory management for temporary and persistent data addressed shortcomings in traditional memory management. This new design involved a spillable page layout, which could be spilled with almost no (de-) serialisation overhead, by lazily recomputing pointers to variable-size data in-place. In this chapter, we integrate these techniques into DuckDB's hash join, which enables it to process larger-than-memory intermediates and effortlessly spill them to storage.

We then tackle a problem that is specific to joins, which have two inputs, unlike the sort and aggregation operators, which have one input. Hash joins are blocking for *only one* of their inputs. This entails that they use memory simultaneously with other active operators. Their combined memory usage cannot exceed the memory limit, as illustrated in Figure 5.1, which requires managing the memory of *all active operators*. Efficiently managing the memory of multiple operators helps ensure multiple hash joins proceed simultaneously, thereby reducing the latency of evaluating the query plan.

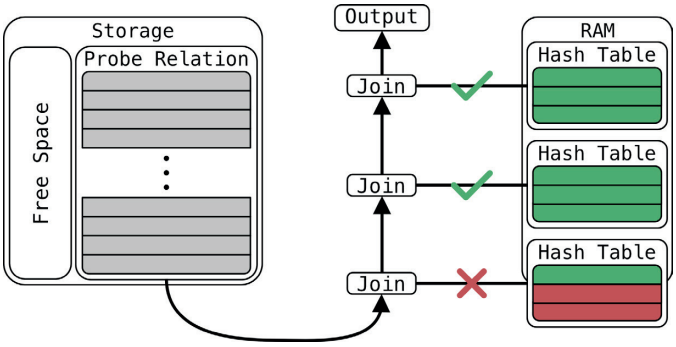

Figure 5.1 A query plan where three hash joins are probed in a single pipeline. The combined space required for the hash tables cannot exceed RAM; therefore, to "save" the hash joins (instead of using sort-merge), the joins must "spill" some data to storage. Query engines must also decide how to distribute RAM over the joins, which affects the amount of data spilled in each join and the total amount of spilled data.

5.1 Introduction

Shortcomings in external query processing are particularly present in analytical systems, which became mainstream after DBMS optimised for main memory, despite OLAP systems frequently processing large volumes of data and having large intermediates. Many analytical systems either abort queries because they do not support larger-than-memory intermediates or switch to a traditional disk-based algorithm that is orders of magnitude slower, introducing a "performance cliff", as demonstrated in Chapter 4. For large grouped aggregations with many unique keys, this cliff can be avoided using adaptive algorithms [31].

[31] Thanh Do et al. (2023), "Efficient Sorting, Duplicate Removal, Grouping, and Aggregation"

For *hash joins*, merely implementing an adaptive algorithm is insufficient to enable robust external query processing. If a single

memory-intensive operator, like grouped aggregation, is evaluated in a query plan, all memory is available to that operator. The *join*, however, has two inputs. One of the hash join's inputs is materialised and could require considerable memory, while the other can usually be streamed to the next operator in the query plan. Therefore, multiple memory-intensive operators can be active simultaneously. We show such a query plan in Figure 5.1. The combined memory usage of active operators cannot exceed the memory limit, and the available RAM must be distributed over the various active operators. Some distributions may under-utilise RAM and over-utilise storage; therefore, efficient distribution is critical to query performance [86, 87, 88].

[86] Biswadeep Nag et al. (1998), "Memory allocation strategies for complex decision support queries"

[87] Benoît Dageville et al. (2002), "SQL Memory Management in Oraclegi"

[88] Josep Aguilar-Saborit et al. (2008), "Exploiting Pipeline Interruptions for Efficient Memory Allocation"

Efficient utilisation of secondary storage is key to providing good performance at a low cost [69], especially given the high bandwidth of solid-state drives available today. If OLAP systems implemented robust external query processing, *i.e.*, adaptive algorithms and memory control of multiple active operators, they could perform analytical workloads on much more economical hardware. This would then reduce the cost of challenging analytical workloads such as large self-joins, join plans including Parquet files, which are difficult to reorder because only lightweight statistics [56] are available, and join queries with strings, which are ubiquitous in real-world data [78] and require more memory than integer keys.

[69] David Lomet (2018), "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed"

[56] Guido Moerkotte (1998), "Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing"

[78] Adrian Vogelsgesang et al. (2018), "Get Real: How Benchmarks Fail to Represent the Real World"

Contributions. This chapter revisits *external joins*, *compressed execution*, and *multi-operator memory control* for modern OLAP systems to enable robust, larger-than-memory analytical join processing. As stated in Chapter 1, the main contributions of this chapter are the following:

- A *parallel* and external hash join that integrates *unified memory management* and the *spillable page layout* for intermediates, which *gracefully degrades performance* as the memory limit is exceeded.
- A dynamic *multi-operator memory control* approach that efficiently distributes the available memory over simultaneously active operators, with a *low synchronisation overhead* to prevent the performance degradation of parallel query execution.
- An optimiser that uses statistics to create expressions to *compress* and *decompress* columns *during execution*, reducing the space requirement of operators without the need to modify them.

We have implemented the proposed techniques in DuckDB [16]. They have been available since version 1.2.0.

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

Outline. The rest of the chapter is organised as follows. Section 5.2 (page 77) discusses related work. After describing how DuckDB manages temporary data in Section 5.3 (page 80), we present our external hash join in Section 5.4 (page 81). We describe our approach to compressed execution in Section 5.5 (page 86) and managing concurrent operators' memory in Section 5.6 (page 87). In Section 5.7

(page 92), we describe our experimental setup, which we use to evaluate and compare our implementation with other systems in Sections 5.8 to 5.10. Finally, we summarise the chapter and discuss future research in Section 5.11 (page 100).

5.2 Related Work

This section discusses related work on larger-than-memory joins, compressed execution, and multi-operator memory control.

Join Algorithms. The accepted approach to external joins in traditional disk-based database systems is to use sort-merge [89], through an external sort implementation [29]. Once RAM prices decreased, the efficiency of hash-based algorithms over sort-based algorithms was quickly recognised [7]. Hash joins have become the most common way to join relational data. However, a simple hash join cannot offload (spill) data to storage. Not all workloads fit in memory; therefore, disk-based algorithms remain necessary.

[89] M. W. Blasgen et al. (1977), "Storage and Access in Relational Data Bases"

[29] Donald E. Knuth (1998), *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*

[7] David J DeWitt et al. (1984), "Implementation Techniques for Main Memory Database Systems"

In most systems, the query planner must try to choose an efficient join algorithm, based on cardinality estimates, which is *risky*. Many systems produce significant estimation errors that grow as the number of joins increases [17]. A poor choice has enormous implications, and in some cases, inserting *a single row* into one of the input tables can determine the choice of a different algorithm. Choosing a simple hash join can cause queries to abort if the hash table does not fit in memory. Choosing sort-merge can cause execution time to increase by orders of magnitude if a hash table fits.

[17] Viktor Leis et al. (2015), "How Good Are Query Optimizers, Really?"

GRACE [32] eliminates this decision with a hash join algorithm that performs reasonably well in main memory while also being able to process more data than fits in RAM, allowing it to perform an external join while avoiding the $O(n \log n)$ time complexity of sorting. GRACE is also parallelisable: partitions are independent; therefore, they can join in parallel. However, skewed partition sizes create load-balancing issues. Alternatively, each thread can build a hash table on some tuples from every partition of the inner relation and probe it with *all* tuples from the outer relation [32]. This approach resists skew but increases the total probing effort.

[32] Masaru Kitsuregawa et al. (1983), "Application of hash to data base machine and its architecture"

[32] Masaru Kitsuregawa et al. (1983), "Application of hash to data base machine and its architecture"

Hybrid Hash Join (HHJ). HHJ [22] improves GRACE hash join by keeping the first partition in memory and using it to perform a simple hash join, only performing GRACE hash join for the other partitions. HHJ equals a simple hash join if all data fits in memory and gracefully degrades to GRACE hash join as the data size exceeds the memory limit. Implementing a single algorithm for in-memory and larger-than-memory joins leads to more robustness: the optimiser no longer has to decide between join algorithms with vastly different performance characteristics.

[22] Leonard D. Shapiro (1986), "Join Processing in Database Systems with Large Main Memories"

Research on HHJ has focused on making the algorithm more adaptive. Instead of statically partitioning, a dynamic strategy can be used [90] that does not depend on statistics and is, therefore, resistant to skew. Recent research [91] explores dynamic partitioning policies for HHJ, which can reduce spilling and, therefore, I/O cost. The Memory-Contention Responsive Hash Join [92] is a variant of HHJ that adapts to fluctuations in memory contention at runtime, *i.e.*, it can increase or decrease its memory usage during execution.

[90] Masaya Nakayama et al. (1988), "Hash-Partitioned Join Method Using Dynamic Destaging Strategy"

[91] Shiva Jahangiri et al. (2022), "Design Trade-Offs for a Robust Dynamic Hybrid Hash Join"

[92] Diane L. Davison et al. (1994), "Memory-Contention Responsive Hash Joins"

Little research has focused on the parallelism of HHJ. Mourad et al. [93] derive a limit on the parallelism that can be realised with HHJ under skew when each thread processes a partition, but the authors offer no improvements to the algorithm itself. HHJ can also execute in parallel using *plan-driven* parallelism, which splits query plans into fragments and connects them through the *exchange* [8] operator, keeping the operator implementation largely *unaware of parallelism*. However, plan-driven parallelism also suffers from skewed data distributions. *Morsel-driven* parallelism [9] addresses many of the problems of plan-driven parallelism and can achieve near-linear speedups with additional CPU cores, but requires operators to be *parallelism-aware*[1].

[93] Antoine N. Mourad et al. (1994), "Limits of parallelism in hash join algorithms"

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

[1] The technique for the *pipelining hash join* algorithm [94] Annita N. Wilschut et al. (1995), "Parallel evaluation of multi-join queries" could potentially be applied to Hybrid Hash Join, but it seems unlikely that it would yield any benefit over a morsel-driven approach.

Streaming Query Execution. G-join [30] is a sort-based join algorithm that performs well for inputs that fit in memory, but adapts to larger-than-memory inputs in a performance-robust way. However, G-join always fully *materialises* the outer relation, even if the inner relation fits in memory; therefore, more memory bandwidth is needed, which can quickly become a performance bottleneck [11], and much of the benefit of streaming query execution is lost.

[30] Goetz Graefe (2012), "New Algorithms for Join and Grouping Operations"

[11] Peter A. Boncz et al. (1999), "Database Architecture Optimized for the New Bottleneck: Memory Access"

Radix hash join [95] is a hardware-conscious join algorithm, popular in literature in the last decade, which also materialises both input relations. However, a thorough evaluation of this algorithm [50] revealed that it only performs better than the simple hash join for rather specific inputs. Materialising the tuples dominates the overall runtime costs, especially for selective joins.

[95] Jens Teubner et al. (2013), "Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware"

[50] Maximilian Bandle et al. (2021), "To Partition, or Not to Partition, That is the Join Question in a Real System"

Compressed Execution. The size of data streamed through non-blocking query operators is low; therefore, compressing it does not meaningfully reduce the memory footprint. However, blocking operators have a much higher memory footprint because they inherently materialise data. *Compressed execution* [96] is a technique that compresses integer and string values in hash tables to reduce the memory footprint. This has been shown to improve the performance of in-memory workloads, and it can most likely improve the performance of larger-than-memory workloads by reducing the amount of data that needs to be spilled to storage.

[96] Tim Gubner et al. (2021), "Optimistically Compressed Hash Tables & Strings in the USSR"

Multi-operator Memory Control. In non-streaming query execution, a single relational operator is active at any given time. In streaming query execution, multiple memory-intensive operators may be active concurrently *within a single query plan* when hash joins are involved. Active operators must be assigned a portion of the memory pool, as their combined memory usage may not exceed the limit. A naive approach would be to reserve a fixed amount of memory for each query and each operator. However, in many cases, this will lead to over- or under-utilisation of memory, namely when the fixed amount is larger or smaller than the required amount. Under-utilisation of memory is especially detrimental for the Hybrid Hash Join, as it reduces the effectiveness of the initial in-memory hash join, thereby requiring more tuples from the (often larger) outer relation to be partitioned and spilled.

Dageville et al. [87] implement a *dynamic* approach to multi-operator memory control that adapts to the workload during execution by applying cost-based rules. A global memory manager publishes a memory bound at three-second intervals, to which operators must promptly react. This approach improves throughput and reduces memory footprint compared to reserving a fixed amount per operator or thread. However, it maintains the same memory limit for all operators in a query plan; therefore, a memory-intensive operator cannot exceed this bound, even if the other operators in the query plan are less memory-intensive. In such cases, query performance suffers due to underutilisation of RAM.

[87] Benoît Dageville et al. (2002), "SQL Memory Management in Oracle9i"

Another dynamic approach to memory control is proposed in [97]. However, this approach adapts to workloads that change over time; therefore, it does not apply to the problem of managing the memory of concurrently active operators within a single plan.

[97] Adam J. Storm et al. (2006), "Adaptive self-tuning memory in DB2"

Aguilar-Saborit et al. [88] remark that dynamic adjustments can be expensive and propose a *static* approach to multi-operator memory control that assigns allocations to operators before execution. Their approach applies to bushy plans and builds on prior work that only considered left-deep query plans [86]. A post-optimisation phase identifies operators in the query plan that are active concurrently and enumerates memory assignments to find a near-optimal solution. This approach allows memory-intensive operators to use more memory than other operators and considers that not all operators are active simultaneously; therefore, it can assign more memory to the active operators, improving RAM utilisation.

[88] Josep Aguilar-Saborit et al. (2008), "Exploiting Pipeline Interruptions for Efficient Memory Allocation"

[86] Biswadeep Nag et al. (1998), "Memory allocation strategies for complex decision support queries"

An obvious shortcoming of static approaches is that they assign memory based on cardinality estimates; therefore, they are prone to over- or underutilisation of RAM due to estimation errors. Another shortcoming of the dynamic and static approaches discussed so far is that they reserve memory for hash joins *too early*. Lang et al. [98] delay allocating a hash table and inserting tuples into it until *after* materialising the inner relation, such that building the hash table can be fully parallelised. If temporary data is spillable, *e.g.*, by using the techniques proposed in Chapter 4, this approach can delay a

[98] Harald Lang et al. (2015), "Massively Parallel NUMA-Aware Hash Joins"

hash join from using memory until right before probing starts: only a minimal amount is needed for materialisation. This leaves more memory available to other operators, which is especially useful in bushy query plans.

Dynamic memory control does not rely on cardinality estimates to assign memory to operators, as it assigns memory based on true cardinalities observed during execution. Dynamically adapting to workloads causes overhead, especially in parallel query execution. In plan-driven parallelism [8], each thread has its own hash join operator; therefore, resizing a hash table does not require cross-thread communication. However, the number of concurrently active operators increases with the number of threads, which complicates memory management on many-core CPU architectures. In morsel-driven parallelism [9], threads share a single hash join operator, which avoids this problem. However, cross-thread communication is required to resize a shared hash table.

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

5.3 Managing Temporary Data

The external hash join that we present in this chapter integrates the techniques for managing temporary query intermediates proposed in Chapter 4. We have improved spilling intermediates with two additional techniques, which we discuss in this section.

Buffer Eviction Policy. After DuckDB materialises temporary data on pages, the pages are unpinned. Unpinning effectively frees up memory, as the buffer manager can offload these pages to storage whenever necessary, allowing for larger-than-memory data materialisation. In DuckDB's partitioned external hash aggregation, described in Section 4.5 (page 59), this approach enables fine-grained spilling of pages instead of coarse-grained approaches that spill entire hash partitions in an all-or-nothing manner. However, a drawback of this approach is that operators no longer have control over which partitions are in memory, unless they are pinned. If the buffer manager arbitrarily offloads pages, it may offload pages that will be reused shortly, causing them to be evicted and reloaded unnecessarily. To prevent *thrashing*, the buffer manager should prioritise offloading pages that will be pinned later in the execution.

To achieve this, we exploit DuckDB's tendency to process hash partitions sequentially, *i.e.*, partition 0, 1, 2, *etc.*, and use this information to prioritise spilling pages from, *e.g.*, partition 42 instead of spilling pages from partition 0[2]. Pages from the same partitions are evicted using a *least-recently-used* policy. Persistent data is always evicted before any temporary data, as evicting persistent data does not require writing to storage (because it is already stored in the database file). No distinction is made between temporary pages belonging to queries from different connections; they are added to the same queue.

[2] This optimisation applies both to the partitioned aggregation presented in Chapter 4, and the partitioned join that we present in this chapter.

Adaptive Compression. DuckDB's page layout is spillable without any serialisation, a design goal motivated by the speed of modern SSDs. However, in some situations, it can be beneficial to compress the pages. This reduces I/O overhead by writing smaller pages, at the cost of spending some CPU time to compress them. If I/O availability is low, *e.g.*, due to many threads writing to storage in parallel, this trade-off is favourable.

DuckDB uses lightweight columnar compression [77] for persistent data, which reduces the size, while incurring less (de-)compression overhead than general-purpose compression algorithms. These columnar compression techniques cannot be applied directly to our spillable page layout, as the layout is row-oriented; therefore, we compress the pages using ZSTD [99], a general-purpose compression algorithm. Compressing is done adaptively, *i.e.*, only when we measure that the time spent compressing and writing a page is lower than the time spent writing an uncompressed page[3], similar to the design proposed by Umami [100].

[77] Marcin Zukowski et al. (2006), "Super-Scalar RAM-CPU Cache Compression"

[99] Yann Collet (2015), *Zstandard - Fast real-time compression algorithm*

[3] This optimisation applies any time DuckDB spills data to external storage, regardless of whether the pages belong to an aggregation, join or even a temporary table.

[100] Maximilian Kuschewski et al. (2024), "High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance"

By integrating our techniques for managing temporary data into DuckDB's hash join, like we did for DuckDB's hash aggregation, the hash join operator can spill data through the buffer manager; therefore, it can process larger-than-memory query intermediates by partitioning the data, with only minor modifications to the in-memory algorithm, as will be explained in detail in the next section.

5.4 External Hash Join

This section first discusses the considerations that went into designing DuckDB's hash join before presenting the implementation.

Considerations. The external processing capabilities of the hash join should not compromise in-memory performance, and performance should degrade gracefully as the size of the temporary intermediates exceeds the memory limit, like with DuckDB's external hash aggregation, as described in Section 4.5 (page 59). This requirement effectively rules out unnecessarily materialising probe-side data, as this significantly degrades in-memory performance [50] and increases the space requirement. Not compromising on in-memory performance also necessitates deferring the decision to perform any external processing to query execution, as this avoids the optimiser erroneously deciding to deviate from the in-memory strategy based on statistics.

[50] Maximilian Bandle et al. (2021), "To Partition, or Not to Partition, That is the Join Question in a Real System"

To achieve graceful degradation as the memory limit is exceeded, the join operator should not have a "*hard switch*" when it deviates from the in-memory strategy at execution time, as this design will cause a "*performance cliff*". Therefore, the join should continue with the in-memory strategy as long as possible, even if the build side of the join exceeds the memory limit. The extent to which the overall strategy would deviate from the in-memory strategy should be proportional to the extent to which the size of the temporary query intermediates

exceeds the memory limit, *i.e.*, a different strategy should only be used for the portion of data that does not fit in memory. HHJ [22] achieves this by performing a simple hash join on as much data as will fit in memory and only performing a partitioned hash join on the amount of data that exceeds the memory limit.

[22] Leonard D. Shapiro (1986), "Join Processing in Database Systems with Large Main Memories"

The implementation should be fully parallel at every step, as any single-threaded execution limits many-core scalability, as per Amdahl's law [5]. The degree of parallelism should be skew-resistant, which rules out parallelisation over hash partitions, as data distributions can skew their sizes, causing some threads to perform much more work than others, like with plan-driven parallelism [8]. Instead, we use the concept of morsel-driven parallelism [9] throughout the entire join, always scheduling tasks over *fine-grained* fragments of input data. Although I/O is likely to be the bottleneck when processing query intermediates larger than RAM, modern storage devices are highly concurrent and should be accessed concurrently whenever possible to achieve maximum throughput.

[5] Gene M. Amdahl (1967), "Validity of the single processor approach to achieving large scale computing capabilities"

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

Implementation. We will now give an overview of DuckDB's hash join implementation, which was designed to fulfill these requirements. DuckDB's in-memory hash join is inspired by HyPer's hash join [9] but extended with key optimisations and larger-than-memory processing capabilities. Our design is illustrated in Figure 5.2. The algorithm has multiple distinct phases, which we explain in detail.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

Building. In the building phase, we assign morsels to threads until all build-side data has been read. There can be many more morsels than threads. Data is scanned from morsels in batches of up to 2,048 tuples, which is DuckDB's standard vector size. The threads hash the join key columns and materialise tuples into partitions that use DuckDB's spillable page layout. As explained in Section 5.3 (page 80), the pages are then unpinned; therefore, they can be evicted by the buffer manager when necessary.

The partitioned data is row-oriented, while the incoming data is column-oriented: the conversion from a column-oriented to a row-oriented layout takes place while partitioning the data. By materialising and partitioning tuples simultaneously, rather than materialising, then partitioning, we avoid copying tuples *more than once*, reducing the partitioning cost. This matters because maintaining the performance of the in-memory hash join is one of our design goals, which does not strictly require partitioning, unlike the external hash join.

Partitions are determined by radix, *i.e.*, a few of the middle bits of the hash. The lower and upper bits of the hash are used to determine the entry in the hash table and improve collision resolution performance, as will be explained. The bits used for the three roles must not overlap, as this makes them less effective. Initially, only four radix bits are used, creating $2^4 = 16$ partitions. After all build-side data has been partitioned, all thread-local data is exchanged to a global state. If any

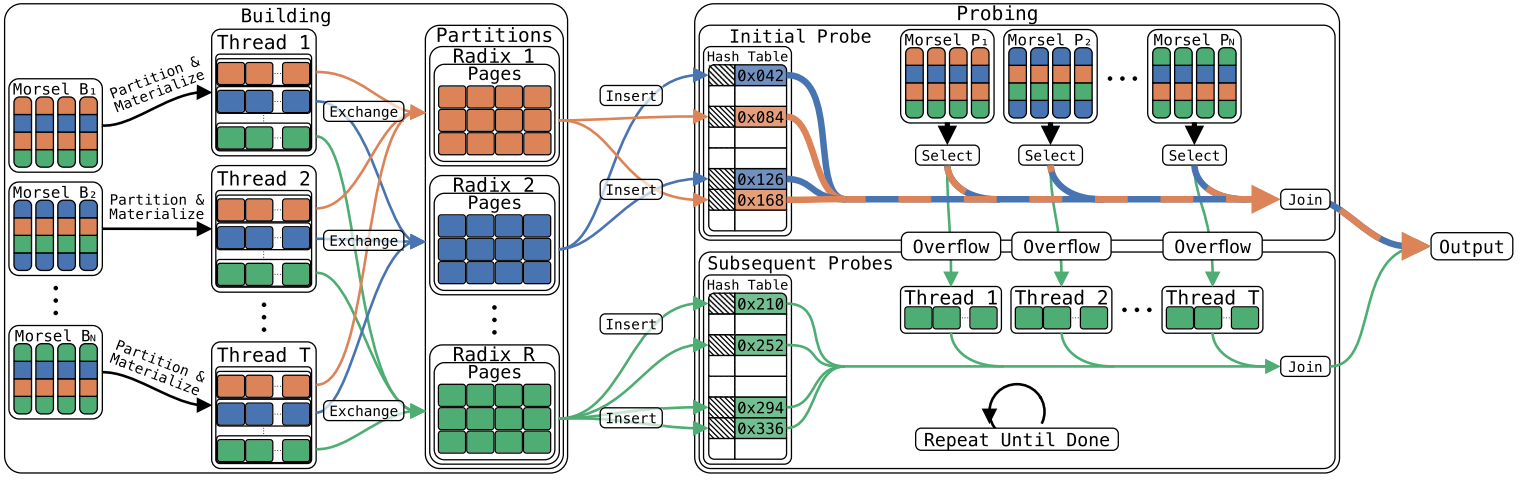

Figure 5.2 DuckDB's hash join. *Building* is done by assigning morsels to threads until all build-side data has been read. Each thread radix partitions and materialises the data to DuckDB's spillable page layout, and a hash table is built on one or more partitions. Then, *probing* starts with an *initial probe*. Morsels are assigned to threads until all probe-side data has been read. Each thread selects the tuples from the data that can join with the partitions in the hash table and probes, and immediately outputs matching tuples. The *overflow* tuples that are not selected are also partitioned and materialised. After the initial probe, all remaining data from both sides has materialised within the join operator, which is processed in a series of *subsequent probes*.

single partition is too large to fit in memory, the data is repartitioned using more radix bits, creating more partitions.

Hash Table Creation. After the building phase, tuples can be inserted into the hash table. We select as many partitions as will fit in memory to insert into the hash table to maximise memory utilisation. If all partitions fit, the join will proceed fully in memory. Checking whether all partitions fit is a simple and inexpensive operation that does not affect in-memory performance.

After selecting the partitions, we allocate and zero-initialise an array of 64-bit entries with a large enough capacity to fit an entry for each tuple in the selected partitions. Then, each thread is assigned morsels from the selected partitions to insert into the array until all tuples have been inserted. The tuples' hashes are scanned, and the offsets into this array are obtained from the lower bits of the hash. The pointers to the tuples are added to the 64-bit entries at these offsets using atomic compare-and-swap operations [98].

[98] Harald Lang et al. (2015), "Massively Parallel NUMA-Aware Hash Joins"

Collision Resolution. Hash collisions, *i.e.*, tuples hashing to the same bucket, are resolved using a *combination of linear probing and chaining*. We reduce the random accesses incurred by linear probing using the upper 16 bits of the hash, which we call *salt*. Pointers have a width of 64 bits on 64-bit CPU architectures, but only the lower 48 bits are used for the address, as this already allows for up to ≈ 281 TB of address space. We store the salt in the remaining 16 bits of the 64-bit entries. When probing, we first compare the salt. If the salt is equal, we follow the pointer to compare the join keys, like in DuckDB's hash aggregation, described in Section 4.5 (page 59).

However, a key difference between the join and aggregate hash tables is that the join hash table may contain multiple tuples with the *exact same* join keys, which would be deduplicated in grouped aggregation. DuckDB's join hash table treats these collisions differently from hash collisions that have *different* join keys.

If we encounter a hash collision (and matching salt) while inserting a tuple into the hash table, we compare its join keys with the join keys of the colliding tuple. If the join keys are not equal, the collision is simply resolved using linear probing. If the join keys are equal, however, we create a *chain* instead, *i.e.*, a linked list of tuples with the exact same join keys. This enables us to create chains that only feature one unique set of join keys.

By performing these comparisons once while building the hash table, we need not perform them while probing (possibly multiple times). If a probe tuple's join keys are equal to those of the first build tuple in a chain, the probe tuple is joined with all build tuples in the chain without further comparisons. This approach reduces the number of performed comparisons and linear probing collisions in many-to-many joins. Furthermore, the chains could also be used for factorisation [67], which is crucial for graph query processing[4]. This

[67] Daniel Flachs et al. (2022), "The 3D Hash Join: Building On Non-Unique Join Attributes"

[4] At the time of writing, factorisation has not been implemented in DuckDB.

optimisation also enables us to detect the opposite case, which is when there are *no duplicate join keys at all*. We use this information to improve performance while probing, as we know that there are no hash collision chains; therefore, we can skip (attempting to) follow the chains. Probing starts after the hash table has been created. We distinguish two probe phases: the *initial* probe and one or more *subsequent partitioned* probes.

Initial Probe. In this phase, we assign morsels to threads until all probe-side data has been read. The join key columns are hashed for every batch of data the threads receive, and the radix is extracted from the hash. Tuples with a radix that matches one of the inserted build-side partitions can probe the hash table immediately. Hence, these tuples are *selected*. This selection does not imply partitioning or materialisation; only the creation of a *selection-vector* [18]. The selected tuples probe the hash table and continue pipelined query execution. All build-side partitions are inserted into the hash table for the in-memory hash join. By default, all probe-side tuples are selected, and their radix does not need to be extracted; therefore, in-memory performance is unaffected.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

Probe-side tuples with a radix that does not match one of the inserted build-side partitions are not selected for external hash joins. These tuples are partitioned and materialised to a spillable page layout. The page layout for the probe-side data is similar to that of the materialised build-side data, as it also uses lazy pointer recomputation, as described in Section 4.4 (page 56). An important difference is that the probe-side data will not be randomly accessed but sequentially scanned; therefore, we do not need a row-oriented layout to improve locality [19]. Instead, the partitioned probe-side data is stored using a column-oriented tuple layout. A column-oriented layout is the default for DuckDB's execution engine, allowing the data to be scanned without copying it.

[19] Marcin Zukowski et al. (2008), "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing"

Subsequent Partitioned Probes. For external hash joins, the initial probe is followed by one or more partitioned probes. First, however, after the initial probe, the hash table may have to be scanned, depending on the *join type*; for example, unmatched tuples are scanned if it is a *right* join. The hash table is scanned in parallel by splitting the data into morsels of, *e.g.*, $\approx$ 100,000 tuples. If all build-side data fits into memory, the join is done after completing this scan. If not, the remaining build- and probe-side data that has been partitioned and materialised still needs to be joined and output by performing one or more rounds of partitioned probes.

Each subsequent partitioned probe cycles through three "stages": ① Inserting build-side tuples into the hash table, ② Probing the hash table, and ③ Scanning the hash table (if the join type requires it). A thread-global state oversees the progress and assigns tasks to the active threads. During this phase, the hash table is again built on as many partitions as possible, allowing for longer uninterrupted

probing, thereby reducing the need for synchronisation. After all partitions have been processed, the join is done.

DuckDB's external hash join, as described here, has been part of every DuckDB release since version 1.1.0[5]. With this design, the decisions regarding the external hash join are made dynamically during execution, without relying on statistics. In every phase of the join, we have been careful not to compromise the in-memory performance with the larger-than-memory functionality. If the build side exceeds the memory limit, memory utilisation is maximised by performing an in-memory hash join on a portion of the data that fits in RAM. Although there are some points where threads must synchronise, *e.g.*, to decide which partitions will be inserted in the hash table, the bulk of the work is fully parallelised using morsel-driven parallelism.

[5] The initial version of DuckDB's external hash join was released in version 0.5.0.

Up to this point in this chapter, we have described how DuckDB's join adheres to a memory limit, but *multiple operators could be active simultaneously*. In Section 5.6 (page 87), we present our approach to multi-operator memory control and explain how it is integrated with the hash join.

5.5 Compressed Materialisation

External joins require materialising data from both relations. If a quarter of the inner relation fits in RAM, three-quarters of the outer relation must be materialised. Most joins are *asymmetric*, with the inner relation often much smaller than the outer relation. The benefit of fitting, *e.g.*, 10% more of the inner relation in memory is enormous, as it avoids materialising another 10% of the potentially much bigger outer relation.

DuckDB has had its *Compressed Materialisation* optimiser since version 0.9.0, which has been integrated into the hash join since version 1.1.0. This optimiser creates projections to compress columns before materialising operators such as joins and projections to decompress the columns afterward. By implementing compressed execution as an optimiser, instead of within hash tables [96], the (de-)compression logic is centralised and applicable to all operators without modification.

[96] Tim Gubner et al. (2021), "Optimistically Compressed Hash Tables & Strings in the USSR"

In systems with columnar storage, the difference between storing an integral column as a 32-bit or 64-bit integer is small because lightweight compression methods [77] such as frame-of-reference and bitpacking compress the data to a similar size. During execution, these columns are usually decompressed to the type as defined in the schema. If the data is streamed, the difference in memory footprint between a 32-bit and a 64-bit integer column is negligible, as only a few tuples are in memory at a time. However, if the data is materialised, *e.g.*, in a hash table, many tuples can be in memory: reducing their size can considerably reduce the memory footprint.

[77] Marcin Zukowski et al. (2006), "Super-Scalar RAM-CPU Cache Compression"

Integer Compression. If min/max statistics are available, they are used to apply frame-of-reference compression to compress a *wide* integer type to a *thinner* integer type. If the *range* (max - min) of an integer column is small enough to fit in a thinner type, we subtract the minimum and cast to the thinner type, reducing its width by at least half. This transformation can be reversed by casting back to the wider type and adding the minimum again. It can be applied to *all* columns from the inner relation, even if the column is used as a join key, because the transformation does not affect comparisons as long as the corresponding column from the outer relation is transformed in the same way.

String Compression. DuckDB uses the optimised 16-byte string layout used by HyPer [26] and Umbra [54]. With this representation, the width of an arbitrarily sized `VARCHAR` is 16 bytes, which includes a pointer if the string is longer than 12 characters. If the maximum string length is known to be, *e.g.*, 3 bytes, we store the string in a 32-bit integer instead, saving 12 bytes per value. When a string is converted to an integer type, its comparison properties can be preserved by copying the bytes in reverse order on little-endian machines. This conversion reduces memory usage and speeds up comparisons, as integer comparisons are much more efficient than string comparisons.

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

These projections are inexpensive, and adding them improves performance substantially because they reduce, for example, hash table memory requirements.

5.6 Multi-Operator Memory Control

To get a better understanding of how operator memory assignment affects performance, we consider an example where an outer relation O joins two inner relations, I_1 and I_2, in a single pipeline, where I_1 is joined first, and then I_2, depicted in Figure 5.3. The pipeline ends in the consumption of the data in operator P, for example, an ungrouped aggregation. We denote the memory limit with L. To simplify our example, we assume that: (1) Scanning O and executing P requires no memory, (2) O is much larger than L, and its join keys are uniformly distributed, (3) Both joins are non-selective one-to-many relationships, *i.e.*, all tuples from O will find exactly one match in both joins. These assumptions allow us to analyse simplified cases where external processing is required.

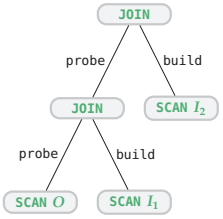

Figure 5.3 Example query plan with two joins for Multi-Operator Memory Control.

Memory Assignments. Consider case (A) where $L = 1\,\text{GB}$, and I_1 and I_2 are 1 GB each. We cannot fully perform both joins in main memory; therefore, we must decide how to divide the memory between them. If we assign the first join with I_1 1 GB and the second join with I_2 nothing, the first join can proceed fully in memory, but the second join does not have any memory to build a hash table; therefore, it cannot perform a streaming probe with *any* data coming

from O. With this assignment, all of O is fully materialised in the second join (with I_2). According to our assumption, O is much larger than L; therefore, spilling it has a considerable I/O.

If we assign both joins 500 MB of memory, both can perform a streaming probe with half of the probe-side data while having to materialise the other half. With the previous 1,000/0 assignment, all probe-side data is materialised once. The 500/500 assignment seems to have the same implications because half the probe-side data is materialised twice, once in each join. However, with the 1,000/0 assignment, the space requirement is twice as large, as the probe-side data is materialised *all at once* in the second join. The 500/500 assignment is preferable because it materialises only up to half of the probe-side data at any given time.

Relation Sizes. The size of the inner relations affects how memory is assigned. Consider again the same example with $L = 1$ GB, but now, for case (B), the inner relations have different sizes: I_1 is 200 MB, and I_2 is 1,800 MB. Choosing for *equality*, *i.e.*, a 500/500 memory assignment, is wasteful as I_1 only needs 200 MB. Choosing for *equity*, *i.e.*, a 100/900 assignment, would again cause half the data to be materialised in both joins. A better assignment is 200/800, as this allows the first join to proceed fully in memory while only requiring $^{10}/_{18}{}^{\text{th}}$ of the probe-side data to be materialised in the second join. So, by materialising just $^{1}/_{18}{}^{\text{th}}$ more of the probe-side data in the second join, we avoid materialising half of O in the first join.

From this example, it seems like more memory should always be given to smaller joins, but this leads to *starvation* of larger joins. Consider case (C), where I_1 is 1 GB and I_2 is 2 GB. We could assign 1 GB to the first join and none to the second to reduce the total amount of materialised data. With this assignment, all data is materialised in the second join, and none of the data is fully streamed through the pipeline into the consuming operator P; therefore, this assignment increases the overall space requirement. To address this complex trade-off, we propose a cost model that can be optimised dynamically during query execution.

Cost Model. The examples above identify two aspects to consider when assigning memory to joins: *materialisation cost*, which should be minimised, and *pipeline throughput*, which should be maximised. As we have seen, these aspects are at odds with each other, as reducing overall materialisation may cause throughput to drop to zero, and increasing throughput may increase materialisation cost. Therefore, we have created a cost model that considers materialisation cost *and* pipeline throughput.

We express a join's *materialisation cost* as the fraction of the probe-side data that must be materialised. This fraction depends on the size s_i of the inner relation I_i and how much memory a_i is assigned to the join. We weigh this fraction by a weight w_i, as each join's materialisation

cost can differ. In DuckDB, w_i is equal to the width of the probe-side tuples, as the size of the materialised data, and, therefore, the cost increases linearly with this width.

We define the total materialisation cost $\mathbf{M}(A, S)$ of a join pipeline that joins an outer relation with N inner relations as the sum of these weighted fractions, where $S = \{s_1, s_2, \ldots, s_N\}$ and $A = \{a_1, a_2, \ldots, a_N\}$. We express a join's *throughput* as the inverse of the materialisation cost: the fraction of the probe-side data that can be streamed through it. This fraction also depends on the operator's size and memory assignment. We define the throughput $\mathbf{T}(S, A)$ of a join pipeline as the geometric mean of these fractions: a value between 0 and 1.

We formally define $\mathbf{M}(A, S)$ and $\mathbf{T}(S, A)$ as

$$\mathbf{M}(S, A) = \sum_{i=1}^{N} w_i \cdot \left(1 - \frac{a_i}{s_i}\right), \qquad \mathbf{T}(S, A) = \left(\prod_{i=1}^{N} \frac{a_i}{s_i}\right)^{\frac{1}{N}}.$$

As mentioned, $\mathbf{M}(S, A)$ must be minimised, whereas $\mathbf{T}(S, A)$ must be maximised. We incorporate both into a single cost model $\mathbf{C}(S, A)$ by multiplying the materialisation cost with (1 - throughput), *i.e.*, $\mathbf{C}(S, A) = \mathbf{M}(S, A) \cdot (1 - \mathbf{T}(S, A))$.

According to this cost model, the best memory assignments for the cases are: (A) 500/500, (B) 200/800, (C) $666.\bar{6}/333.\bar{3}$, assuming widths w_i are equal for each join. The cost model assigns memory equally when the relations are of equal size and prioritises smaller joins when they are not. It does so without *starving larger joins completely*, as starving any join causes $\mathbf{T}(S, A)$ to quickly become 0, which increases the overall cost due to the multiplication with $1 - \mathbf{T}(S, A)$ in the cost model.

The model is sufficiently complex to express the combination of the materialisation cost and pipeline throughput into a single number. It expresses materialisation cost and throughput pipeline only using relative, not absolute, sizes; therefore, the cost is the same if constant factors scale the assignments, sizes, and memory limit. The inputs to the proposed cost model are *observed values*, not estimated ones, *i.e.*, accurate memory usage is obtained during execution. The model is also not unnecessarily complex; for example, it does not attempt to account for variables such as estimated selectivity or memory and storage bandwidths. Such variables are not guaranteed to remain constant over time; therefore, if inaccurate, they could negatively impact the cost model.

Static vs. Dynamic. As discussed in Section 5.2 (page 77), static approaches to multi-operator memory control [86, 88] rely on statistics, which are not always present. Even when present, statistics are prone to producing unreliable cardinality estimates for join order optimisation [17], which can produce plans with much larger intermediates and are, therefore, orders of magnitude worse than optimal. If relied upon for multi-operator memory control, these cardinality estimation errors could cause memory to be assigned inefficiently, spilling more data and considerably degrading performance.

[86] Biswadeep Nag et al. (1998), "Memory allocation strategies for complex decision support queries"

[88] Josep Aguilar-Saborit et al. (2008), "Exploiting Pipeline Interruptions for Efficient Memory Allocation"

[17] Viktor Leis et al. (2015), "How Good Are Query Optimizers, Really?"

We opt for a fully *dynamic* approach where memory control is interleaved with query execution, observing the actual cardinalities in the pipelines as they occur and reacting accordingly. However, unlike the dynamic approach proposed by Dageville et al. [87], we avoid limiting operators' memory usage, as this can cause memory to be underutilised, as discussed in Section 5.2 (page 77). Instead, we allow large joins to use more memory, if necessary, like the static approach of Aguilar-Saborit et al. [88].

[87] Benoît Dageville et al. (2002), "SQL Memory Management in Oracle9i"

[88] Josep Aguilar-Saborit et al. (2008), "Exploiting Pipeline Interruptions for Efficient Memory Allocation"

Dynamically adjusting memory while executing a query can be expensive, because hash tables may have to be *resized*. This is especially costly if systems have a high degree of execution parallelism, as threads may have to wait while the hash table is being resized. We must be careful not to impair parallel performance, especially in the typical case where all operators fit in RAM and extensive control of the memory of multiple operators is not needed. Our approach must, therefore, have a *low synchronisation overhead*.

Dageville et al. [87] use a background thread to broadcast memory assignments at a fixed interval, to which operators must react. Their system uses plan-driven parallelism [8], in which operator parallelism is *encapsulated*. In this framework, each thread has its own plan fragment with a copy of each operator. This complicates dynamic multi-operator memory control with high degrees of parallelism, as there are many active operators whose memory usage can be adjusted and whose combined memory usage must not exceed the limit.

[87] Benoît Dageville et al. (2002), "SQL Memory Management in Oracle9i"

[8] Goetz Graefe (1990), "Encapsulation of Parallelism in the Volcano Query Processing System"

DuckDB uses morsel-driven parallelism [9], in which operators are parallelism-aware. In this framework, there is only one copy of each operator on which threads work in parallel. There are *fixed synchronisation points*, which demarcate when operators start and end for all participating threads. This design simplifies dynamic multi-operator memory control, as it is much easier to get an overview of the memory usage of operators, pipelines, and query plans as a whole.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

Dynamic Assignments. In DuckDB, operators indicate the amount of data accumulated at these fixed synchronisation points and receive a memory assignment in return. For our external hash join, this happens right before a hash table is built from materialised data, *i.e.*, before the initial probe and each subsequent partitioned probe. This reduces the communication between operators and the data structure responsible for assigning memory to operators, which we call the *Temporary Memory Manager* (TMM).

At first glance, this approach may not seem flexible enough to produce efficient memory assignments. In our running example, the hash table for the join with I_1 may be built first. Some memory would be assigned to this join without considering the join with I_2. The memory assignment of the join with I_1 can only be reduced at this join's own request. This leaves less memory available for the join with I_2, with no way to adapt.

As explained in Section 5.4 (page 81), DuckDB first performs *Build-Side Materialisation*, in which all build-side data is materialised using a spillable page layout. Large amounts of data can be materialised without pinning pages in memory because the buffer manager can spill them. Only after all build-side data has been materialised, DuckDB performs *Hash Table Creation*, in which tuples are inserted into a hash table, which finally requires data to be in memory.

We exploit the delay between *Build-Side Materialisation* and *Hash Table Creation* to compute efficient memory assignments at runtime. Hash table construction for simultaneously active joins is postponed until the data for *all* such joins has been materialised. In our example, I_1 and I_2 are first materialised, after which their sizes are communicated to the TMM. Then, both join operators request memory assignments and build their hash tables accordingly. By deferring these requests until all relations have materialised, the cost model can fairly divide the memory between the operators, as it has all the needed information to optimise the cost model.

This approach ensures that only the join operators that are probed in the same pipeline request memory simultaneously. This reduces the problem of assigning memory to active operators to left-deep plans as in [86], effectively eliminating the need to consider bushy plans for memory assignment [88].

[86] Biswadeep Nag et al. (1998), "Memory allocation strategies for complex decision support queries"

[88] Josep Aguilar-Saborit et al. (2008), "Exploiting Pipeline Interruptions for Efficient Memory Allocation"

Cost Model Optimisation. When a memory assignment is finally requested, parameter S is known, *i.e.*, the array of sizes of the materialised relations of the cost model $\mathbf{C}(S, A)$. The memory assignments, parameter A, are to be decided by the TMM. We approximate the optimal (according to the cost model) memory assignments with a few iterations of gradient descent. First, we initialise each $a_i \in A$ with a minimum assignment that the operator needs. Then, in each iteration, we compute the gradient of $\mathbf{C}(S, A)$ with respect to each a_i. We increase the a_i with the lowest gradient by a fraction of the free memory. Note, however, that, unlike regular gradient descent, this increase is bounded, as the memory assignment a_i of the operator i should not exceed its total size s_i.

To reduce the potential cost of this optimisation, we perform a fixed number of iterations of gradient descent, and only a few times per operator, at the aforementioned fixed synchronisation points, *e.g.*, before the initial probe and each subsequent probe in the external hash join. This also only happens when query execution can benefit from it, *i.e.*, if the total size of the relations exceeds the memory limit; therefore, in-memory performance is unaffected. The time spent optimising the cost model is small compared to the time spent processing the larger-than-memory data.

Concurrency and Ownership. In DuckDB, there is one *shared* TMM for all active connections, meaning that the TMM is also responsible for managing the memory of concurrent queries. If multiple

connections are concurrently executing join pipelines, the TMM optimises the cost model as if the operators from all connections *belong to the same pipeline*. The memory that the TMM assigns to operators is *not owned* by the TMM but by the operators themselves. The TMM only returns a limit to an operator that requests an assignment, which the operator *promises* to never exceed.

The concurrency and ownership design of the TMM in DuckDB is an *implementation decision*. If an individual memory limit per connection must be imposed, one TMM should be used per connection instead of a shared TMM for all active connections. Similarly, if centralised ownership of memory allocations is desired, the TMM could *own* the memory it assigns to operators.

The TMM has been implemented in DuckDB since version 0.10.0 and has since been integrated into the hash join and aggregation operators. The TMM, as described here, was released in version 1.1.0[6].

[6] The initial version of the TMM did not optimise a cost model; it merely ensured that the combined memory usage of active operators did not exceed the memory limit.

5.7 Experimental Setup

Our experiments run on AWS EC2 using Ubuntu 24.04. We run each query 5 times and report the median execution time. If any of the 5 runs does not complete within 1,000 seconds, we report a timeout.

Hardware. We choose the `c6id.4xlarge` AWS instance type for our experiments, which is the exact same instance as described in Section 4.6 (page 65). This instance has an Intel Xeon 8375C CPU with 8 cores (16 threads) and 32 GB of DDR4 RAM. The instance has access to 1 TB of NVMe Instance Storage, which is physically attached to the host, unlike the usual Elastic Block Storage [101] on EC2. We set the instance's tenancy to `dedicated` so that the entire node is reserved, but we do not use the rest of the node's capacity. This setup eliminates the *noisy neighbor* problem that cloud environments may exhibit; therefore, our results are consistent.

[101] Amazon Web Services (2021), *Amazon Elastic Block Store*

Data Generation. We generate data using different parameter configurations, creating a variety of larger-than-memory joins, allowing us to evaluate how these parameters affect performance.

We consider the following four parameters: ① *Cardinality*, the number of rows. We vary this parameter between 100 million and 1,000 million. ② *Width*, the number of *sets* of projected columns. A set of columns consists of one *tag* column (a single uppercase letter), one *employee* column (of length 15, *e.g.*, `EMPN04242424242`), and one *comment* column (a sentence consisting of four random lorem ipsum words, with an average length of around 27 characters). We vary this parameter between 0 and 4. We only use string columns because they are ubiquitous in real-world data [78] and more challenging to process than fixed-width columns. ③ *Skew*, the skewness of the join keys. The skewness is determined by sampling from a power law

[78] Adrian Vogelsgesang et al. (2018), "Get Real: How Benchmarks Fail to Represent the Real World"

distribution with parameter α. We vary α between 0, for which the distribution is uniform, and 1, for which the distribution is Zipfian, and approximately 75% of join keys are identical. ④ *Unique Key Count*, the number of unique join keys *from which to sample*. We vary this parameter between 40 million and 200 million.

Isolating Performance. While large join pipelines can easily dominate a query's runtime, *isolating* the performance of internal components of database systems is difficult because we can often only reliably observe end-to-end query runtime. Some systems have built-in query profilers, but these differ from system to system; therefore, their measurements cannot create a fair comparison. The end-to-end runtime includes unwanted and unrelated overheads, such as query planning and transferring the result set through a client protocol. Client protocol serialisation is especially costly and can dominate query execution time [43]. We must reduce the impact of these overheads to measure the performance of joins reliably.

[43] Mark Raasveldt et al. (2017), "Don't Hold My Data Hostage: A Case for Client Protocol Redesign"

With sufficiently large workloads, any planning overhead becomes insignificant. To reduce transfer protocol overhead, we add an ungrouped aggregation to the query, reducing the result set size to one row. The aggregation must be cheap so that it does not affect execution time. One of the cheapest aggregate functions would be `COUNT(col)`, but this can be replaced with `COUNT(*)` by the optimiser if `col` does not contain any `NULL` values. This would allow the optimiser to remove that column entirely from the query, reducing the overall memory usage. Instead, we use the `ANY_VALUE(col)` aggregate function, which is a cheap aggregate function that returns an arbitrary value from the input. In theory, this yields the same result as adding a `LIMIT 1` clause to the query. *However*, adding `LIMIT 1` allows most systems to terminate queries early after seeing one row. The `ANY_VALUE(col)` approach causes the systems below to fully evaluate the workloads while also yielding a result set of just one row.

Systems. We compare DuckDB [16] (version 1.2.0), against one traditional database system and two systems with strong analytical performance. Traditional systems are often orders of magnitude slower than modern OLAP systems at analytical workloads for various reasons, such as large amounts of per-tuple overhead [18]. However, they offer robust external query processing; therefore, they are interesting to compare against.

[16] Mark Raasveldt et al. (2019), "DuckDB: An Embeddable Analytical Database"

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

Besides DuckDB, the systems under evaluation are the following:

PostgreSQL [6] (version 16.3) The popular, open-source, traditional, disk-based relational DBMS for OLTP workloads, initially developed at UC Berkeley.

[6] Michael Stonebraker et al. (1986), "The design of POSTGRES"

HyPer [26] (version 2023.3), a main-memory-based relational DBMS for mixed OLTP and OLAP workloads, which uses data-centric code generation, developed at Technische Universität München (TUM), now Tableau's data engine.

[26] Thomas Neumann (2011), "Efficiently Compiling Efficient Query Plans for Modern Hardware"

[54] Thomas Neumann et al. (2020), "Umbra: A Disk-Based System with In-Memory Performance"

[84] Jason Evans (2006), "A scalable concurrent malloc (3) implementation for FreeBSD"

[7] DuckDB uses `jemalloc`, a general-purpose, high-performance `malloc` implementation that reduces memory fragmentation, to allocate memory.

[17] Viktor Leis et al. (2015), "How Good Are Query Optimizers, Really?"

[8] We do not include ClickHouse and MonetDB due to poor performance.

[9] Experiments: https://github.com/lnkuiper/experiments/tree/master/oochj, archived at https://doi.org/10.5281/zenodo.15967009.

Umbra [54] (v0.1 2024-04-17), HyPer's successor, a "Disk-Based System with In-Memory Performance" that also uses data-centric code generation, developed at TUM.

For DuckDB, we use `SET allocator_background_threads=true;`, which enables background threads through DuckDB's `jemalloc` [84] extension[7] to speed up allocation performance. For HyPer and Umbra, we use default settings. For PostgreSQL, we use similar settings to those used by Leis et al. [17], and set `work_mem` to 1 GB, `shared_buffers` to 4 GB, and `effective_cache_size` to 30 GB. We also set `temp_file_limit` to -1, to allow unlimited spilling[8]. All of our experiments are publicly available on GitHub[9]

5.8 Evaluation: External Join

In this section, we experimentally evaluate and compare our external join implementation with other implementations. We perform an inner join of two tables, evaluating only the external join. Multi-operator memory control will be evaluated in Section 5.9 (page 96).

Workload. In our experiments, we change one of the data generation parameters described in Section 5.7 (page 92) at a time, allowing us to isolate its effect on external joins. We first establish a *default* parameter configuration as a starting point. The default inner relation has a cardinality of 200M rows, a width of 1 set of columns, no skew (*i.e.*, $\alpha = 0$), and 200M unique keys. With these parameters, the size of the inner relation is roughly equal to the available memory in our experimental setup, although due to internal differences, this size is not equal for all systems. DuckDB can fit approx. $^2/_3{}^{\text{rd}}$ of the inner relation in RAM with this configuration. The default outer relation has a cardinality of 500M rows, a width of 1 set of columns, skewed with $\alpha = 0.5$, and 200M unique keys. We create six experiments by varying the inner and outer relations' cardinality, width, and skew.

When varying the cardinality of the inner relation, we ensure that all tuples will be matched by setting the number of unique keys *equal* to the cardinality of the inner relation. Increasing the cardinality/width of the inner and outer relations stresses the ability of systems to join larger volumes of data. Execution time should increase linearly with the cardinality/width of the relations. However, algorithm choice and spilling may introduce non-linearity in systems, which this experiment should reveal.

When varying α of the inner relation, we keep α of the outer relation fixed at 0, and vice versa, to avoid joining two skewed distributions together and creating a join that is close to a cross product. For the skew experiments, we expect execution time to decrease slightly for higher α because the random access from probing the hash table will become skewed towards join keys that appear more often, which reduces overall cache misses. For highly skewed data distributions with $\alpha = 1$ (Zipfian), approximately 75% of the data belongs to a

single partition. These experiments should reveal whether systems are skew-resistant, *e.g.*, if they use partitions to parallelise or spill entire partitions in an all-or-nothing manner.

For our final experiment, we vary the unique key count. With lower key counts, tuples can have multiple matches, and the result cardinality is larger than either input relation. For example, with an inner relation cardinality of 200M and a unique key count of 40M, the result cardinality is 5× the cardinality of the outer relation, assuming all tuples find a match. We expect execution time to increase with fewer unique keys, as the join will emit more tuples. This experiment should reveal how well systems deal with *exploding joins*, which may cause issues such as many hash collisions.

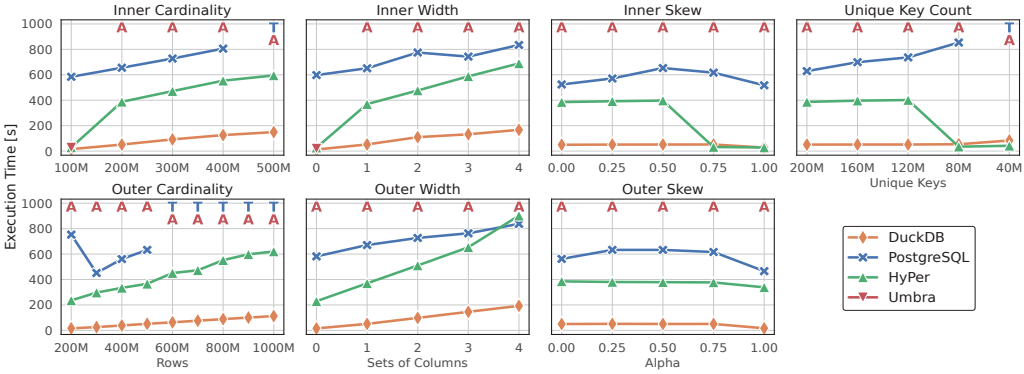

Figure 5.4 Execution times for external joins with different data generation parameter configurations. Lower is better. T denotes that the query timed out after 1,000 seconds. A denotes that the query was aborted due to running out of memory.

Results. We show the results in Figure 5.4. In the *Inner Cardinality* experiment, execution times increase linearly as expected, except HyPer when going from 100M to 200M rows due to switching the join algorithm. At 100M rows, all but PostgreSQL perform an in-memory join. PostgreSQL has an execution time of 584*s* for 100M and times out at 500M rows. From 100M to 200M rows, DuckDB's execution time increases from approx. 15*s* to 51*s*, HyPer's from approx. 29*s* to 388*s*, and Umbra's from roughly 28*s* to aborting. The *Inner Width* experiment tells a similar story with expected linear scaling, although PostgreSQL manages to finish within the timeout. When going from 0 to 1 set of columns, DuckDB's execution time increases from approx. 12*s* to 52*s*, HyPer's from approx. 27*s* to 370*s*, and Umbra's from approx. 28*s* to aborting. Umbra is unable to finish any other query.

In the *Outer Cardinality* experiment, PostgreSQL chooses a poor algorithm for 200M rows but scales linearly afterward. From 600M rows onward, at least one of the five runs takes more than 1,000s for PostgreSQL. DuckDB and HyPer scale linearly as expected, although HyPer is orders of magnitude slower due to using a sort-merge join. In the *Outer Width* experiment, all systems show linear scaling as expected. PostgreSQL and HyPer barely finish with the timeout with 4 sets of columns, however, taking approx. 838*s* and 902*s*, respectively, while DuckDB takes approx. 193*s*.

In the *Inner Skew* experiment, DuckDB's execution time is consistent at approx. 51*s*, although this decreases to 28*s* for $\alpha = 1$ for which the distribution is Zipfian. PostgreSQL's execution time is mostly consistent, between 518*s* and 653*s*. HyPer's execution time drops from 397*s* to approx. 32*s* when going from $\alpha = 0.5$ to $\alpha = 0.75$ because a hash table suddenly fits in main memory due to fewer unique keys. The *Outer Skew* experiment shows similar behaviour, although HyPer's execution time is consistent. When going from $\alpha = 0.75$ to $\alpha = 1$, DuckDB's execution time decreases from approx. 51*s* to 16*s*, PostgreSQL's from 616*s* to 466*s*, and HyPer's from 377*s* to 339*s*. Many sorting algorithms exploit skew; therefore, PostgreSQL's and HyPer's sort-merge join also benefit.

Note that DuckDB would not have been able to finish joins with larger Zipfian inner relations, as the largest hash partition must fit in RAM. DuckDB should always be able to finish joins with large Zipfian outer relations, as partitions from this side do not need to fit fully in RAM because they are read in a streaming fashion.

Finally, in the *Unique Key Count* experiment, DuckDB's execution time increases from approx. 51*s* for 200M unique keys to 83*s* for 40M unique keys, as expected due to the increased number of emitted tuples. The same applies to PostgreSQL, which times out at 40M unique keys. Similar to the Inner Skew experiment, when going from 120M to 80M unique keys, HyPer switches from its external to its in-memory join due to fewer unique keys, causing execution time to decrease from 401*s* to approx. 35*s*.

5.9 Evaluation: Pipelined External Joins

In this section, we experimentally evaluate multi-operator memory control with workloads in which multiple joins are processed simultaneously in a pipeline. We exclude the other systems, as the previous section showed that DuckDB is the only system under benchmark that can complete larger-than-memory joins in a (partially) streaming fashion. When systems opt for an external sorting approach, only one operator is evaluated simultaneously, and multi-operator memory control becomes irrelevant.

Policies. We compare four different memory assignment *policies* in DuckDB: ① *WeightedCost*: the cost model proposed in Section 5.6 (page 87). ② *UnweightedCost*: equivalent to *WeightedCost* with weight w_i always equal to 1. ③ *Equality*: statically assigns each operator $1/N^{\text{th}}$ of the total available memory, where N is the total number of operators. ④ *Equity*: dynamically assigns operator j with observed size s_j an amount of memory equal to s_j divided by S_{ALL} of the total available memory, where S_{ALL} is the sum of all operator sizes, *i.e.*, Equity assigns joins memory proportional to their size.

We expect Weighted- and UnweightedCost to perform similarly. However, the width of the outer relation increases after every join by gath-

ering columns from the inner relation. This should give WeightedCost an edge, as it considers this when optimising the cost model. Equality and Equity should perform well when processing joins with similar sizes but poorly when they have different sizes, especially Equity, as assigning small joins less memory should increase materialisation cost, as explained in Section 5.6 (page 87).

Workload. We create nine interesting join pipeline scenarios that allow us to evaluate how the different policies behave. We only consider left-deep query plans, as only these joins are active simultaneously in DuckDB, as explained in Section 5.6 (page 87). In these query plans, there is one outer relation and one or more inner relations. We use the same parameter configuration for the outer relation here as in the previous section: a cardinality of 500M rows, a width of 1 set of columns, skewed with $\alpha = 0.5$, and 200M unique keys. For our inner relation(s), we use a similar parameter configuration as before: a width of 1 set of columns, no skew (*i.e.*, $\alpha = 0$). The inner relation(s) have a varying number of rows, and the number of unique keys is always equal to the number of rows.

Our first (baseline) scenario is denoted $s40$, which is a join between the outer relation and one inner relation with 400M rows. The inner relation's size is roughly equal to $2\times$ the available memory in our experimental setup. The second scenario is a join with two inner relations, each with a cardinality of 200M rows, which we denote with $s20/20$, *i.e.*, *we denote scenarios with the cardinalities of the inner relations* that are probed in the same pipeline. We list all nine scenarios in Table 5.1. Relations are probed in the encoded order, *i.e.*, sX/Y probes X first, then Y.

To facilitate a comparison between different scenarios, we require as an *invariant* that the sum of the cardinalities of the inner relation(s) is the same in each of the scenarios, which results in a similar space requirement for each scenario. We also use `LEFT` joins to prevent the outer relation's cardinality from getting reduced by joins, as this would affect all subsequent joins in the pipeline and make it difficult to compare scenarios with each other. We create seven more scenarios, for a total of nine scenarios, representing a wide variety of left-deep plans that could occur in challenging join queries. These will be explained alongside the results.

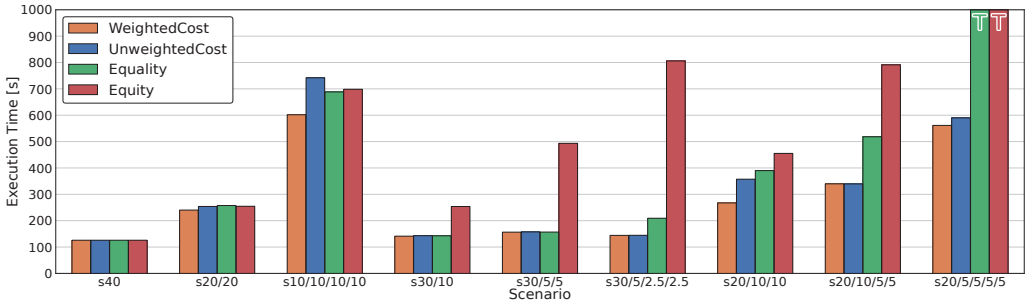

Figure 5.5 Execution times for join pipeline scenarios with different memory assignment policies. Lower is better. 'T' denotes that the query timed out after 1,000 seconds.

Table 5.1 Join pipeline scenarios.

Scenario	Description
*s*40	Baseline
*s*20/20	2 equi-sized
*s*10/10/10/10	3 equi-sized
*s*30/10	1 large, 1 small
*s*30/5/5	1 large, 2 small
*s*30/5/2.5/2.5	1 large, 3 small
*s*20/10/10	1 medium, 2 small
*s*20/10/5/5	1 medium, 3 small
*s*20/5/5/5/5	1 medium, 4 small

Results. We show the results in Figure 5.5. The first three scenarios, *s*40, *s*20/20, and *s*10/10/10/10, have equi-sized inner relations. There is no small join that should be prioritised in RAM. Instead, the best strategy is to assign a similar amount of memory to each of the joins, which all four policies do. WeightedCost has a slight edge over the other three policies, as the width of the outer relation increases after every join due to gathering the columns from the inner relation, which it takes into account by assigning slightly more memory to joins that appear later in the pipeline.

For the next three scenarios, *s*30/10, *s*30/5/5, and *s*30/5/2.5/2.5, the large 300M join should be assigned less memory to allow the much smaller inner relations of $\leq$100M to fit in RAM. This is exactly what Weighted- and UnweightedCost do. As a result, their execution times here are only slightly higher than for *s*40. Equality also does this for *s*30/10 and *s*30/5/5, but not for *s*30/5/2.5/2.5, as it causes the 50M join to spill, degrading performance unnecessarily. As expected, Equity performs worse with each added join due to its memory assignments, causing each join in the pipeline to spill.

The last three scenarios, *s*20/10/10, *s*20/10/5/5, and *s*20/5/5/5/5, have one fairly large 200M join that should be assigned less memory than the smaller joins of 100M or 50M. However, the size of the smaller joins totals 200M, meaning that they cannot all fit in RAM; therefore, assigning memory is much less straightforward than for the previous three scenarios. Equity assigns little memory to smaller joins; therefore, it again performs worse with each added join, eventually timing out. For *s*20/10/10, the best strategy is to perform as much of the two 100M joins in memory. Equality assigns more memory to the 200M join than Weighted- and UnweightedCost and finishes in 390*s*. UnweightedCost does this slightly better and finishes in 357*s*, while WeightedCost finishes in 267*s*. For *s*20/10/5/5, the best strategy is to perform both 50M joins in memory and as much of the 100M join as possible, leaving some space for the 200M join, which is exactly what Weighted- and UnweightedCost do. Equality takes 180*s* longer, as it assigns less memory to the 100M join, causing more data to be spilled. Finally, for *s*20/5/5/5/5, Equality assigns slightly less memory to each 50M join than Weighted- and UnweightedCost; therefore, each of the four joins spills more, causing a timeout.

As expected, Weighted- and UnweightedCost perform better than the more naive policies, with WeightedCost being slightly better on some queries. Although Equality and Equity lead to worse results, they are much better than having *no policy at all*, as this would result in running out of memory and aborting the query or having to resort to processing one operator at a time, *e.g.*, with sort-merge joins.

5.10 Evaluation: TPC-H

In this section, we experimentally evaluate overall external query processing capabilities using the analytical TPC-H benchmark at SF 1,000, again using the hardware described in Section 5.7 (page 92).

We exclude PostgreSQL from this benchmark, as it suffers from large amounts of per-tuple overhead [18] and does not unnest the correlated subqueries in TPC-H [102], causing quadratic performance. Furthermore, PostgreSQL does not compress the database file sufficiently to fit within the limited storage space available on the EC2 instance type. We also exclude Umbra from this benchmark, as the evaluation in Section 5.8 (page 94) shows that it cannot process larger-than-memory intermediates. Therefore, only DuckDB and HyPer are evaluated. Both implement full unnesting of arbitrary queries and have similar query plans for most queries.

[18] Peter A. Boncz et al. (2005), "MonetDB/X100: Hyper-Pipelining Query Execution"

[102] Thomas Neumann et al. (2015), "Unnesting Arbitrary Queries"

Results. Figure 5.6 shows the results of the benchmark. DuckDB and HyPer perform similarly for all but five queries: 7, 9, 10, 13, and 18. HyPer switches to out-of-core processing for these queries and is orders of magnitude slower than DuckDB. If we exclude these queries, the longest-running query for both systems is query 21, which takes 183s for DuckDB and 250s for HyPer. If we include these queries, DuckDB's longest-running query is Q9, which takes 241s, while HyPer takes 875s for Q13, and times out after 1,000 seconds for queries 7, 9, 10, and 18. Only queries 9, 13, and 18 require out-of-core processing for DuckDB, and despite spilling to storage, their execution time is not much higher.

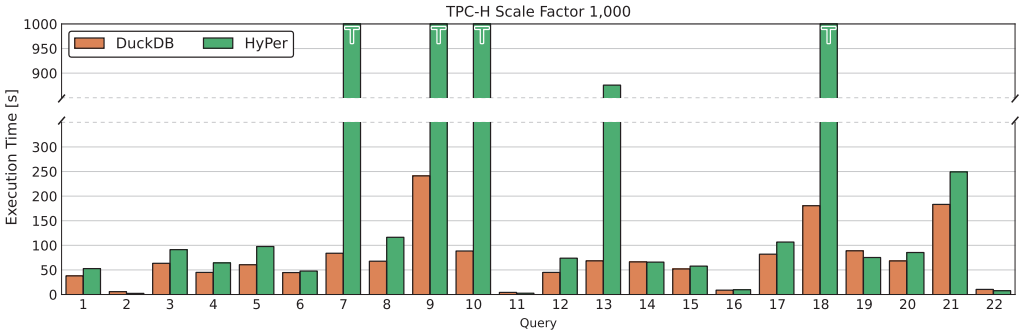

Figure 5.6 Execution times for TPC-H queries 1 through 22 at scale factor 1,000 (split y-axis). Lower is better. 'T' denotes that the query timed out after 1,000 seconds.

Join-Heavy Queries. Queries 7, 9, and 10 have memory-intensive joins that dominate the execution time. DuckDB filters out data early in Q7 because it implements the "Join-Dependent Expression Filter Pushdown" query optimisation technique [80], which HyPer does not. DuckDB and HyPer generate almost identical query plans for Q9 and Q10. In both plans, multiple joins are probed within a single pipeline, requiring the systems to control the memory of multiple operators. Despite having similar plans, the system's execution times are drastically different. This difference can be attributed to HyPer switching to disk-based algorithms, which potentially causes it to sort the data multiple times. Meanwhile, DuckDB continues to use its hash join.

[80] Peter Boncz et al. (2013), "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark"

Aggregation-Heavy Queries. Queries 13 and 18 have a large, memory-intensive grouped aggregation. The plans for these queries are the same for both systems. The large aggregation is active simultaneously with a join in both queries, complicating memory control. HyPer has strong in-memory aggregation [9] but is forced to switch to a disk-based algorithm due to running out of memory. DuckDB has a similar aggregation strategy but can spill intermediates to disk; therefore, it can complete much larger aggregations with many unique groups without falling back to solutions based on sorting.

[9] Viktor Leis et al. (2014), "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age"

5.11 Summary

In this chapter, we presented three techniques for external join query processing in analytical database systems, which we implemented and released in DuckDB. We experimentally evaluated our implementation and compared it with other systems. Our external join experiments showed that DuckDB could perform external joins much more efficiently and robustly than the other systems. When intermediates fit in memory, DuckDB competes with state-of-the-art systems, showing that our external query processing techniques do not sacrifice in-memory performance. Our pipelined external join experiments showed that DuckDB's dynamic approach to multi-operator memory control assigns memory more efficiently than static or naive approaches.

Our experimental results with TPC-H at scale factor 1,000 showed that our techniques generalise to analytical workloads. DuckDB demonstrates similar execution times for queries, regardless of whether it spills intermediates to storage, showing that larger-than-memory query processing does not have to be slow. This enables larger datasets to be processed on more economical hardware, using less energy than, for example, distributed data management systems.

Future Work. Although DuckDB's hash join performs well on skewed data, *it cannot complete joins* if any of the inner relations' partitions are larger than the memory limit. If many tuples have the same join key, *repartitioning will not help*. After the initial probe, all remaining data has been materialised. Here, DuckDB could decide to *swap the inner and outer relation* depending on the sizes of the materialised partitions. Adaptively swapping sides would reduce the join's reliance on the optimiser even further and allow it to complete skewed joins that it would otherwise not be able to.

Reducing the size of intermediates, especially strings, because they are ubiquitous in real-world workloads, would improve scaling further. Retaining lightweight string compression such as FSST [103] during execution could help reduce the size of hash tables even further. Retaining FSST would require holding onto some of the storage metadata during query execution and decompressing the strings as late as possible, *e.g.*, when used in a comparison expression.

[103] Peter Boncz et al. (2020), "FSST: fast random access string compression"

6. Conclusion

The primary goal of this thesis was to design physical query execution such that query performance degrades gracefully as the execution engine transitions from processing in-memory intermediates to larger-than-memory intermediates, with the added requirement that in-memory performance should not be sacrificed to achieve this goal. This goal arose from the observation that, since the inception of main memory database systems, external query processing has been orders of magnitude slower than in-memory query processing, and never caught up despite considerable advancements in modern storage performance over the past decades.

We identified several shortcomings in how database systems manage memory for intermediates. Due to these limitations, systems could not use available storage resources as a natural extension of memory for temporary data, unlike persistent data. Physical operator implementations incurred substantial overhead when spilling intermediates, or even resorted to entirely different algorithms with different performance characteristics for external processing, eliminating the possibility of graceful performance degradation. This thesis demonstrates how these shortcomings can be overcome with a combination of newly developed[1] and existing[2] techniques. We integrated these techniques into several physical operators and demonstrated that this allowed them to transition from in-memory to external query processing with low overhead. By first approaching this problem from the perspective of in-memory performance, our external query processing techniques have not sacrificed in-memory performance, as shown empirically in comparison to other systems.

[1] Such as *Unified Memory Management*, a *spillable page layout* for temporary query intermediates with *lazy in-place pointer recomputation*, and a novel approach to *dynamic multi-operator memory control*.

[2] Such as a *row-oriented data representation* for improved data locality, and hash partitioning as a *divide-and-conquer* strategy for external aggregation and join operators.

Our memory management techniques do not assume OLAP/OLTP or any particular execution model (*e.g.*, vectorised or compiled); therefore, they can be implemented in any system that supports paged memory. However, large page sizes (larger than the 4 or 8 KiB that are common in traditional OLTP systems) and modifications to paged memory management are necessary. Since paged memory management is a critical component of database systems, we do not expect traditional OLTP systems to adopt our techniques. However, modern systems that wish to utilise available resources better should adopt techniques similar to ours. There has been a recent interest in this area in the research community [104], where an OLTP-focused academic research system has experimented with similar techniques to improve resource utilisation in cloud environments, where fluctuating resource availability and pricing also demand adaptivity.

[104] Riki Otaki et al. (2025), "Resource-Adaptive Query Execution with Paged Memory Management"

Much database research is implemented in academic prototypes only[3]. Implementing all of our research in DuckDB, a widely used and well-tested system, is a major achievement that shows that our techniques are viable in real-world applications. Implementing everything in DuckDB created additional challenges, but it has also allowed us to collect and incorporate user feedback, which has improved our implementation and made it more robust to various workloads. The author of this thesis is still actively contributing to DuckDB and maintaining the code produced for the research presented here.

[3] It can take a long time, if ever, for research to be implemented in a real database system.

While the research presented in this thesis has focused on single-node, single-connection analytical use cases, we believe that our research presents an important contribution to query processing and the field of database systems as a whole.

6.1 Future Work

Parallel K-Way Sorting. The classical approach to external sorting is k-way merging using a tournament tree [29]. This approach incurs little data movement, as only one merge step merges all sorted runs. Furthermore, the final sorted data need not be materialised, as the tuple at the top of the tournament tree can immediately be output in a streaming fashion. The 2-way cascaded merge that we implemented in Chapter 3 incurs much more data movement, as there are $\log k$ merge steps, and the sorted data must be materialised at every step. Despite the obvious drawbacks, we opted for the 2-way cascaded merge because sorting must be parallel to achieve good in-memory performance. To our knowledge, no algorithm exists to parallelise k-way merging in a skew-resistant manner. We parallelised 2-way merging using Merge Path [46].

[29] Donald E. Knuth (1998), *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*

[46] Oded Green et al. (2014), "Merge Path - A Visually Intuitive Approach to Parallel Merging"

Since then, we have discovered that Merge Path can be generalised to k sorted runs. For 2-way Merge Path, partition boundaries are computed using binary search; therefore, its complexity is $O(\log m)$ with m the partition size. For k-way Merge Path, computing partition boundaries requires searching through all k runs simultaneously; therefore, its complexity is $O(k \log m)$. Since k-way merging needs just a single step to merge all sorted runs, this cost is incurred only n/m times *in total*. For 2-way merging, the $O(\log m)$ cost is incurred n/m times *for each of the* $O(\log k)$ *merge steps*, which implies that the cost of computing partition boundaries for k-way merging is *only slightly higher* than it is for 2-way merging[4]. However, this is not an issue, as both costs pale compared to the $O(n \log n)$ needed to perform the sorting.

[4] Written out in full, the complexity of k-way Merge Path is $O(n/m\ k \log m)$, which is slightly higher than the complexity of 2-way Merge Path, which is $O(n/m \log k \log m)$.

We have implemented this fully parallel k-way merge in DuckDB (pull request #17584 [105]), which now integrates the improved page layout proposed in Section 4.7 (page 67) instead of the initial page layout proposed in Section 3.8 (page 45). Output data from the sort operator is materialised *in parallel while preserving the output order*, which DuckDB realises using *batch indices* [106]. The new sorting implementation, which generalises Merge Path to k sorted runs, enables fully parallel k-way merging, thereby allowing a single implementation to achieve high-performance in-memory, and efficient external sorting with minimal data movement and a low memory requirement[5]. This has yet to be integrated into sort-based operators such as sort-merge join, but we believe that this sorting implementation will bring the performance of sort-based external query operators closer to hash-based ones.

[105] Laurens Kuiper (2025), *New Sorting Implementation*

[106] Mark Raasveldt (2022), *Support Parallel Order-Preserving Result Set Materialization*

[5] Unlike the implementation presented in Chapter 3.

Spillable Aggregate States. During grouped aggregation, tuples are matched using their *group keys*, and typically, one or more *aggregate functions* are computed. In Chapter 4, we focused on spilling the group keys, not the aggregated values.

Aggregate functions use a *state* that is updated for every tuple. For the `avg` aggregate function, for example, this contains a *sum* and a *count*. For many aggregate functions, including `avg`, this state has a *fixed size*. Fixed-size aggregate states can be *inlined* in the fixed-size rows in our spillable page layout, allowing them to be spilled to storage. Some aggregate functions, *e.g.*, `string_agg`, and `list`, have a variable size, as each tuple that updates them increases their size. Therefore, these aggregate states cannot be inlined in the rows, *i.e.*, another mechanism is needed to be able to spill them. It would be interesting to create a generalised spilling mechanism for all variable-sized aggregate states to be able to spill any aggregate function.

Responsive Memory Management. The approach to multi-operator memory control proposed in Chapter 5 is flexible enough to manage join query plans of any shape, as it delays assigning memory to operators until the space requirements are known for all joins that are active simultaneously in the query plan. For single-connection use cases, all joins that are active simultaneously belong to the same query plan. For multi-connection use cases, however, new queries could start while other queries are already running. Our approach cannot change the memory usage of joins *while they are active*; therefore, the new queries would either be assigned little memory, causing them to slow down, or the system would have to impose a memory limit per query. Neither approach is attractive, as the former could potentially slow down queries that should be inexpensive, and the latter would lead to over- or underutilisation of memory, similar to how imposing a memory limit per query operator would.

Finding a way to respond to changes in memory contention such that the overall throughput of concurrent queries is maximised would be an interesting direction for future research. One of the main challenges would be to do this without too much synchronisation overhead, which requires a solution with responsive operators [92].

[92] Diane L. Davison et al. (1994), "Memory-Contention Responsive Hash Joins"

Asynchronous I/O. DuckDB uses *synchronous* I/O to read and write data, which entails that threads performing I/O must wait until the I/O operation is complete before they continue processing. However, performing I/O does not require much CPU utilisation because the storage device performs most of the work; therefore, synchronous I/O can lead to low CPU utilisation. For storage devices with high bandwidths, it is beneficial to use *asynchronous* I/O instead, where I/O operations are performed by separate threads that only process I/O operations. Offloading these operations allows the original threads to continue processing without waiting for I/O operations, thereby achieving higher overall CPU utilisation. A recent paper explores asynchronous I/O on a high-bandwidth storage device. It achieves

near-in-memory performance when processing larger-than-memory data, using a spilling approach similar to ours, implemented in an academic prototype system [100]. Further research in this direction with real systems is needed to improve resource utilisation, given the ever-increasing bandwidths of modern storage devices.

[100] Maximilian Kuschewski et al. (2024), "High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance"

6.2 Only Time Will Tell

Traditional database systems implemented external query processing using sort-based operators and were optimised for single-threaded, slow disk access. At the time, this was the best approach to external query processing. This approach has not withstood the test of time, as it does not utilise modern hardware well, *i.e.*, many-core CPUs, and highly parallel SSDs. The techniques presented in this thesis substantially improve the performance of external query processing on modern hardware. It remains to be seen whether our techniques will withstand the test of time, *i.e.*, the future hardware developments.

The current developments, as discussed in Chapter 1, of increased popularity of ARM CPU architecture and a relatively large increase in the I/O speeds and parallelism of modern storage devices, do not affect our techniques. DuckDB has been extensively tested on both x86 and ARM and performs even better on ARM for various workloads. Our techniques should also perform well on modern storage devices with increased I/O speeds and parallelism, although *asynchronous I/O* may become more relevant as this trend continues.

Another trend that has been going on for decades is increased memory availability. Main memory is becoming cheaper and smaller: the 2025 MacBook Pro can get up to 128 GB of RAM. As this trend continues, our techniques may become less relevant, as more data can fit in main memory, and fewer workloads will require external processing. Still, some workloads will exceed any practical RAM capacity. In such cases, our techniques will continue to allow users to robustly process larger-than-memory workloads.

Unforeseen developments could make our techniques obsolete. Our techniques rely on the division of byte-addressable memory and block-addressable storage in the memory hierarchy. If technologies like persistent memory, also discussed in Chapter 1, became more affordable, improve their random access latency, and fully replace SSDs, this division would cease to exist. At the time of writing, however, this seems unlikely.

Whatever the future holds, we hope this thesis inspires researchers to continue innovating database systems to keep up with future hardware developments.

References

[1] E. F. Codd. “A relational model of data for large shared data banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. issn: 0001-0782. url: https://doi.org/10.1145/362384.362685.

[2] Peter J. Denning. “Effects of scheduling on file memory operations”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 9–21. isbn: 9781450378956. url: https://doi.org/10.1145/1465482.1465485.

[3] Donald D. Chamberlin and Raymond F. Boyce. “SEQUEL: A structured English query language”. In: *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’74. Ann Arbor, Michigan: Association for Computing Machinery, 1974, pp. 249–264. isbn: 9781450374156. url: https://doi.org/10.1145/800296.811515.

[4] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade and V. Watson. “System R: Relational Approach to Database Management”. In: *ACM Trans. Database Syst.* 1.2 (June 1976), pp. 97–137. issn: 0362-5915. url: https://doi.org/10.1145/320455.320457.

[5] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS ’67 (Spring). Atlantic City, New Jersey: Association for Computing Machinery, 1967, pp. 483–485. isbn: 9781450378956. url: https://doi.org/10.1145/1465482.1465560.

[6] Michael Stonebraker and Lawrence A. Rowe. “The design of POSTGRES”. In: *Proceedings of SIGMOD 1986*. SIGMOD ’86. Washington, D.C., USA: ACM, 1986, pp. 340–355. isbn: 0897911911. url: https://doi.org/10.1145/16894.16888.

[7] David J DeWitt, Randy H Katz, Frank Olken, Leonard D Shapiro, Michael R Stonebraker and David A. Wood. “Implementation Techniques for Main Memory Database Systems”. In: *Proceedings of SIGMOD 1984*. SIGMOD ’84. Boston, Massachusetts: ACM, 1984, pp. 1–8. isbn: 0897911288. url: https://doi.org/10.1145/602259.602261.

[8] Goetz Graefe. “Encapsulation of Parallelism in the Volcano Query Processing System”. In: *Proceedings of SIGMOD 1990*. SIGMOD ’90. Atlantic City, New Jersey, USA: ACM, 1990, pp. 102–111. isbn: 0897913655. url: https://doi.org/10.1145/93597.98720.

[9] Viktor Leis, Peter Boncz, Alfons Kemper and Thomas Neumann. “Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age”. In: *Proceedings of SIGMOD 2014*. SIGMOD ’14. Snowbird, Utah, USA: ACM, 2014, pp. 743–754. isbn: 9781450323765. url: https://doi.org/10.1145/2588555.2610507.

[10] Emily Blem, Jaikrishnan Menon, Thiruvengadam Vijayaraghavan and Karthikeyan Sankaralingam. “ISA Wars: Understanding the Relevance of ISA being RISC or CISC to Performance, Power, and Energy on Modern Architectures”. In: *ACM Trans. Comput. Syst.* 33.1 (Mar. 2015). issn: 0734-2071. url: https://doi.org/10.1145/2699682.

[11] Peter A. Boncz, Stefan Manegold and Martin L. Kersten. “Database Architecture Optimized for the New Bottleneck: Memory Access”. In: *Proceedings of VLDB 25*. VLDB ’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 54–65. isbn: 1558606157. url: https://doi.org/10.5555/645925.671364.

[12] H. Garcia-Molina and K. Salem. “Main Memory Database Systems: An Overview”. In: *IEEE Trans. on Knowl. and Data Eng.* 4.6 (Dec. 1992), pp. 509–516. issn: 1041-4347. url: https://doi.org/10.1109/69.180602.

[13] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim and Sang-Woo Kim. "A Case for Flash Memory SSD in Enterprise Database Applications". In: *Proceedings of SIGMOD 2008*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 1075–1086. isbn: 9781605581026. url: https://doi.org/10.1145/1376616.1376723.

[14] Maximilian Böther, Otto Kißig, Lawrence Benson and Tilmann Rabl. "Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs". In: *Proceedings of the 17th International Workshop on Data Management on New Hardware*. DAMON '21. Virtual Event, China: Association for Computing Machinery, 2021. isbn: 9781450385565. url: https://doi.org/10.1145/3465998.3466010.

[15] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann and Alfons Kemper. "Persistent Memory I/O Primitives". In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. DaMoN'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. isbn: 9781450368018. url: https://doi.org/10.1145/3329785.3329930.

[16] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An Embeddable Analytical Database". In: *Proceedings of SIGMOD 2019*. SIGMOD '19. Amsterdam, Netherlands: ACM, 2019, pp. 1981–1984. isbn: 9781450356435. url: https://doi.org/10.1145/3299869.3320212.

[17] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *Proc. VLDB Endow.* 9.3 (Nov. 2015), pp. 204–215. issn: 2150-8097. url: https://doi.org/10.14778/2850583.2850594.

[18] Peter A. Boncz, Marcin Zukowski and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *Proceedings of CIDR 2005*. Asilomar, CA, USA: www.cidrdb.org, 2005, pp. 225–237. url: http://cidrdb.org/cidr2005/papers/P19.pdf.

[19] Marcin Zukowski, Niels Nes and Peter Boncz. "DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing". In: *Proceedings of DaMoN 4*. DaMoN '08. Vancouver, Canada: ACM, 2008, pp. 47–54. isbn: 9781605581842. url: https://doi.org/10.1145/1457150.1457160.

[20] Laurens Kuiper, Mark Raasveldt and Hannes Mühleisen. "Efficient External Sorting in DuckDB". In: *Proceedings of BICOD 2021*. Vol. 3163. CEUR Workshop Proceedings. CEUR-WS.org, 2021, pp. 40–45. url: https://ceur-ws.org/Vol-3163/BICOD21%5C_paper%5C_9.pdf.

[21] Laurens Kuiper and Hannes Mühleisen. "These Rows Are Made for Sorting and That's Just What We'll Do". In: *Proceedings of ICDE 39*. Anaheim, California: IEEE, 2023, pp. 2050–2062. isbn: 9798350322279. url: https://doi.org/10.1109/ICDE55515.2023.00159.

[22] Leonard D. Shapiro. "Join Processing in Database Systems with Large Main Memories". In: *ACM Trans. Database Syst.* 11.3 (Aug. 1986), pp. 239–264. issn: 0362-5915. url: https://doi.org/10.1145/6314.6315.

[23] Laurens Kuiper, Peter Boncz and Hannes Mühleisen. "Robust External Hash Aggregation in the Solid State Age". In: *Proceedings of ICDE 40*. New York, NY, USA: IEEE, 2024, pp. 3753–3766.

[24] Laurens Kuiper, Paul Groß, Peter Boncz and Hannes Mühleisen. "Saving Private Hash Join". In: *Proc. VLDB Endow.* 18.8 (Mar. 2025), pp. 2748–2760. issn: 2150-8097. url: https://doi.org/10.14778/3742728.3742762.

[25] G. Graefe. "Volcano-An Extensible and Parallel Query Evaluation System". In: *IEEE Trans. on Knowl. and Data Eng.* 6.1 (Feb. 1994), pp. 120–135. issn: 1041-4347. url: https://doi.org/10.1109/69.273032.

[26] Thomas Neumann. "Efficiently Compiling Efficient Query Plans for Modern Hardware". In: *Proc. VLDB Endow.* 4.9 (June 2011), pp. 539–550. issn: 2150-8097. url: https://doi.org/10.14778/2002938.2002940.

[27] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo and Peter Boncz. "Everything You Always Wanted to Know about Compiled and Vectorized Queries but Were Afraid to Ask". In: *Proc. VLDB Endow.* 11.13 (Sept. 2018), pp. 2209–2222. issn: 2150-8097. url: https://doi.org/10.14778/3275366.3284966.

[28] Cleve Moler. "Matrix computation on distributed memory multiprocessors". In: *Hypercube Multiprocessors* 86.181-195 (1986), p. 31.

[29] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998. isbn: 0201896850.

[30] Goetz Graefe. "New Algorithms for Join and Grouping Operations". In: *Comput. Sci.* 27.1 (Feb. 2012), pp. 3–27. issn: 1865-2034. url: https://doi.org/10.1007/s00450-011-0186-9.

[31] Thanh Do, Goetz Graefe and Jeffrey Naughton. "Efficient Sorting, Duplicate Removal, Grouping, and Aggregation". In: *ACM Trans. Database Syst.* 47.4 (Jan. 2023). issn: 0362-5915. url: https://doi.org/10.1145/3568027.

[32] Masaru Kitsuregawa, Hidehiko Tanaka and Tohru Moto-Oka. "Application of hash to data base machine and its architecture". In: *New Generation Computing* 1.1 (Mar. 1983), pp. 63–74. issn: 1882-7055. url: https://doi.org/10.1007/BF03037022.

[33] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Proceedings of OSDI 6*. OSDI'04. San Francisco, CA: USENIX Association, 2004, p. 10.

[34] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker and Ion Stoica. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of NSDI 9*. NSDI'12. San Jose, CA: USENIX Association, 2012, p. 2.

[35] Frank McSherry, Michael Isard and Derek G. Murray. "Scalability! but at what cost?" In: *Proceedings of HOTOS 15*. HOTOS'15. Switzerland: USENIX Association, 2015, p. 14.

[36] Michael Stonebraker and Andrew Pavlo. "What Goes Around Comes Around... And Around..." In: *SIGMOD Rec.* 53.2 (July 2024), pp. 21–37. issn: 0163-5808. url: https://doi.org/10.1145/3685980.3685984.

[37] Jackie Fenn and Marcus Blosch. *Understanding Gartner's Hype Cycles*. Aug. 2018. url: %7Bhttps://www.gartner.com/en/documents/3887767%7D (visited on 26/11/2024).

[38] Jordan Tigani. *Big Data Is Dead*. Feb. 2023. url: https://motherduck.com/blog/big-data-is-dead/ (visited on 25/11/2024).

[39] Jordan Tigani. *Redshift Files: The Hunt for Big Data*. Aug. 2024. url: https://motherduck.com/blog/redshift-files-hunt-for-big-data/ (visited on 25/11/2024).

[40] George Fraser. *How do people use Snowflake and Redshift?* Sept. 2024. url: https://www.fivetran.com/blog/how-do-people-use-snowflake-and-redshift (visited on 25/11/2024).

[41] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf and Tim Kraska. "Why TPC is Not Enough: An Analysis of the Amazon Redshift Fleet". In: *Proc. VLDB Endow.* 17.11 (Aug. 2024), pp. 3694–3706. issn: 2150-8097. url: https://doi.org/10.14778/3681954.3682031.

[42] Deepak Vohra. "Apache Parquet". In: *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*. Berkeley, CA: Apress, 2016, pp. 325–335. isbn: 978-1-4842-2199-0. url: https://doi.org/10.1007/978-1-4842-2199-0_8.

[43] Mark Raasveldt and Hannes Mühleisen. "Don't Hold My Data Hostage: A Case for Client Protocol Redesign". In: *Proc. VLDB Endow.* 10.10 (June 2017), pp. 1022–1033. issn: 2150-8097. url: https://doi.org/10.14778/3115404.3115408.

[44] Anthony LaMarca and Richard E Ladner. “The Influence of Caches on the Performance of Sorting”. In: *Journal of Algorithms* 31.1 (1999), pp. 66–104. issn: 0196-6774. url: https://www.sciencedirect.com/science/article/pii/S0196677498909853.

[45] Stefan Edelkamp and Armin Weiß. “BlockQuicksort: Avoiding Branch Mispredictions in Quicksort”. In: *ACM J. Exp. Algorithmics* 24 (Jan. 2019). issn: 1084-6654. url: https://doi.org/10.1145/3274660.

[46] Oded Green, Saher Odeh and Yitzhak Birk. “Merge Path - A Visually Intuitive Approach to Parallel Merging”. In: *CoRR* abs/1406.2628 (2014). arXiv: 1406.2628. url: http://arxiv.org/abs/1406.2628.

[47] Orson R. L. Peters. “Pattern-defeating Quicksort”. In: *CoRR* abs/2106.05123 (2021). arXiv: 2106.05123. url: https://arxiv.org/abs/2106.05123.

[48] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 1891–1904. issn: 2150-8097. url: https://doi.org/10.14778/3407790.3407797.

[49] Zuhair Khayyat, William Lucia, Meghna Singh, Mourad Ouzzani, Paolo Papotti, Jorge-Arnulfo Quiané-Ruiz, Nan Tang and Panos Kalnis. “Lightning Fast and Space Efficient Inequality Joins”. In: *Proc. VLDB Endow.* 8.13 (Sept. 2015), pp. 2074–2085. issn: 2150-8097. url: https://doi.org/10.14778/2831360.2831362.

[50] Maximilian Bandle, Jana Giceva and Thomas Neumann. “To Partition, or Not to Partition, That is the Join Question in a Real System”. In: *Proceedings of SIGMOD 2021*. SIGMOD ’21. Virtual Event, China: ACM, 2021, pp. 168–180. isbn: 9781450383431. url: https://doi.org/10.1145/3448016.3452831.

[51] Goetz Graefe. “Implementing Sorting in Database Systems”. In: *ACM Comput. Surv.* 38.3 (Sept. 2006), 10–es. issn: 0360-0300. url: https://doi.org/10.1145/1132960.1132964.

[52] Baktagul Imasheva, Azamat Nakispekov, Andrey Sidelkovskaya and Ainur Sidelkovskiy. “The Practice of Moving to Big Data on the Case of the NoSQL Database, ClickHouse”. In: *WCGO 2019*. Vol. 991. Advances in Intelligent Systems and Computing. Metz, France: Springer, 2019, pp. 820–828. url: https://doi.org/10.1007/978-3-030-21803-4%5C_82.

[53] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender and Martin Kersten. “MonetDB: Two Decades of Research in Column-oriented Database Architectures”. In: *IEEE Data Eng. Bull.* 35.1 (2012), pp. 40–45. url: http://sites.computer.org/debull/A12mar/monetdb.pdf.

[54] Thomas Neumann and Michael J. Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *Proceedings of CIDR 10*. Amsterdam, Netherlands: www.cidrdb.org, 2020. url: http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf.

[55] Daniel Lemire and Owen Kaser. “Reordering Columns for Smaller Indexes”. In: *Inf. Sci.* 181.12 (June 2011), pp. 2550–2570. issn: 0020-0255. url: https://doi.org/10.1016/j.ins.2011.02.002.

[56] Guido Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *Proceedings of VLDB 24*. VLDB ’98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 476–487. isbn: 1558605665.

[57] Anastassia Ailamaki, David J. DeWitt and Mark D. Hill. “Data Page Layouts for Relational Databases on Deep Memory Hierarchies”. In: *The VLDB Journal* 11.3 (Jan. 2002), pp. 198–215. issn: 1066-8888. url: https://doi.org/10.1007/s00778-002-0074-9.

[58] David R. Musser. “Introspective Sorting and Selection Algorithms”. In: *Softw. Pract. Exper.* 27.8 (Aug. 1997), pp. 983–993. issn: 0038-0644.

[59] Juliusz Sompolski, Marcin Zukowski and Peter Boncz. "Vectorization vs. Compilation in Query Execution". In: *Proceedings of DaMoN 7*. DaMoN '11. Athens, Greece: Association for Computing Machinery, 2011, pp. 33–40. isbn: 9781450306584. url: https://doi.org/10.1145/1995441.1995446.

[60] Yihong Zhao, Prasad M. Deshpande and Jeffrey F. Naughton. "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates". In: *SIGMOD Rec.* 26.2 (June 1997), pp. 159–170. issn: 0163-5808. url: https://doi.org/10.1145/253262.253288.

[61] Michael W. Blasgen, Richard G. Casey and Kapali P. Eswaran. "An Encoding Method for Multifield Sorting and Indexing". In: *Commun. ACM* 20.11 (Nov. 1977), pp. 874–878. issn: 0001-0782. url: https://doi.org/10.1145/359863.359892.

[62] Meikel Poess, Bryan Smith, Lubor Kollar and Paul Larson. "TPC-DS, Taking Decision Support Benchmarking to the next Level". In: *Proceedings of SIGMOD 2002*. SIGMOD '02. Madison, Wisconsin: ACM, 2002, pp. 582–587. isbn: 1581134975. url: https://doi.org/10.1145/564691.564759.

[63] Maksim Kita. *JIT in ClickHouse*. Nov. 2022. url: https://clickhouse.com/blog/clickhouse-just-in-time-compiler-jit (visited on 15/01/2026).

[64] Wes McKinney et al. "Data structures for statistical computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.

[65] Richard D Hipp. *SQLite*. Version 3.36.0. 2021. url: https://www.sqlite.org/index.html.

[66] 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". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. url: https://doi.org/10.1038/s41586-020-2649-2.

[67] Daniel Flachs, Magnus Müller and Guido Moerkotte. "The 3D Hash Join: Building On Non-Unique Join Attributes". In: *Proceedings of CIDR 19*. Chaminade: CIDR, 2022, pp. 1–9. url: https://www.vldb.org/cidrdb/papers/2022/p18-flachs.pdf.

[68] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. issn: 0360-0300. url: https://doi.org/10.1145/356770.356776.

[69] David Lomet. "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed". In: *Proceedings of DaMoN 14*. DaMoN '18. Houston, Texas: ACM, 2018. isbn: 9781450358538. url: https://doi.org/10.1145/3211922.3211927.

[70] Viktor Leis, Michael Haubenschild, Alfons Kemper and Thomas Neumann. "LeanStore: In-Memory Data Management beyond Main Memory". In: *Proceedings of ICDE 34*. IEEE Computer Society, 2018, pp. 185–196. url: https://doi.org/10.1109/ICDE.2018.00026.

[71] Andrew Crotty, Viktor Leis and Andrew Pavlo. "Are You Sure You Want to Use MMAP in Your Database Management System". In: *Proceedings of CIDR 19*. 2022. url: https://db.cs.cmu.edu/papers/2022/p13-crotty.pdf.

[72] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck and Christian Dietrich. "Virtual-Memory Assisted Buffer Management". In: *Proc. ACM Manag. Data* 1.1 (May 2023). url: https://doi.org/10.1145/3588687.

[73] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle and Kai-Uwe Sattler. "Cooperative Memory Management for Table and Temporary Data". In: *Proceedings of SiMoD 2023*. SiMoD '23. Bellevue, WA, USA: ACM, 2023. isbn: 9798400707834. url: https://doi.org/10.1145/3596225.3596230.

[74] Goetz Graefe, Wey Guy, Harumi Anne Kuno and Glenn Paullley. "Robust Query Processing (Dagstuhl Seminar 12321)". In: *Dagstuhl Reports* 2.8 (2012), pp. 1–15. issn: 2192-5283. url: http://drops.dagstuhl.de/opus/volltexte/2012/3754.

[75] Felix Martin Schuhknecht, Jens Dittrich and Ankur Sharma. "RUMA Has It: Rewired User-Space Memory Access is Possible!" In: *Proc. VLDB Endow.* 9.10 (June 2016), pp. 768–779. issn: 2150-8097. url: https://doi.org/10.14778/2977797.2977803.

[76] Dominik Durner, Viktor Leis and Thomas Neumann. "On the Impact of Memory Allocation on High-Performance Query Processing". In: *Proceedings of DaMoN 15*. DaMoN'19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. isbn: 9781450368018. url: https://doi.org/10.1145/3329785.3329918.

[77] Marcin Zukowski, Sandor Heman, Niels Nes and Peter Boncz. "Super-Scalar RAM-CPU Cache Compression". In: *Proceedings of ICDE 22*. ICDE '06. USA: IEEE Computer Society, 2006, p. 59. isbn: 0769525709. url: https://doi.org/10.1109/ICDE.2006.150.

[78] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann and Manuel Then. "Get Real: How Benchmarks Fail to Represent the Real World". In: *Proceedings of DBTest 2018*. DBTest'18. Houston, TX, USA: ACM, 2018. isbn: 9781450358262. url: https://doi.org/10.1145/3209950.3209952.

[79] Wes McKinney. *Introducing Apache Arrow Flight: A Framework for Fast Data Transport*. 2019. url: https://web.archive.org/web/20230324080635/https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight.

[80] Peter Boncz, Thomas Neumann and Orri Erling. "TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark". In: *Proceedings of TPCTC 5*. Vol. 8391. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 61–76. isbn: 9783319049359. url: https://doi.org/10.1007/978-3-319-04936-6_5.

[81] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm and Liping Zhang. "DB2 with BLU Acceleration: So Much More than Just a Column Store". In: *Proc. VLDB Endow.* 6.11 (Aug. 2013), pp. 1080–1091. issn: 2150-8097. url: https://doi.org/10.14778/2536222.2536233.

[82] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. "Access Path Selection in a Relational Database Management System". In: *Proceedings of SIGMOD 1979*. SIGMOD '79. Boston, Massachusetts: ACM, 1979, pp. 23–34. isbn: 089791001X. url: https://doi.org/10.1145/582095.582099.

[83] Matteo Frigo, Charles E. Leiserson, Harald Prokop and Sridhar Ramachandran. "Cache-Oblivious Algorithms". In: *ACM Trans. Algorithms* 8.1 (Jan. 2012). issn: 1549-6325. url: https://doi.org/10.1145/2071379.2071383.

[84] Jason Evans. "A scalable concurrent malloc (3) implementation for FreeBSD". In: *Proceedings of BSDCan*. Ottawa, Canada: University of Ottawa, 2006, p. 14.

[85] Philip J. Fleming and John J. Wallace. "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results". In: *Commun. ACM* 29.3 (Mar. 1986), pp. 218–221. issn: 0001-0782. url: https://doi.org/10.1145/5666.5673.

[86] Biswadeep Nag and David J. DeWitt. "Memory allocation strategies for complex decision support queries". In: *Proceedings of CIKM 7*. CIKM '98. Bethesda, Maryland, USA: ACM, 1998, pp. 116–123. isbn: 1581130619. url: https://doi.org/10.1145/288627.288647.

[87] Benoît Dageville and Mohamed Zait. "SQL Memory Management in Oracle9i". In: *Proceedings of VLDB 28*. VLDB '02. Hong Kong, China: VLDB Endowment, 2002, pp. 962–973. url: https://doi.org/10.14778/3007263.3007280.

[88] Josep Aguilar-Saborit, Mohammad Jalali, Dave Sharpe and Victor Muntés Mulero. “Exploiting Pipeline Interruptions for Efficient Memory Allocation”. In: *Proceedings of CIKM 17*. CIKM ’08. Napa Valley, California, USA: ACM, 2008, pp. 639–648. isbn: 9781595939913. url: https://doi.org/10.1145/1458082.1458169.

[89] M. W. Blasgen and K. P. Eswaran. “Storage and Access in Relational Data Bases”. In: *IBM Syst. J.* 16.4 (Dec. 1977), pp. 363–377. issn: 0018-8670. url: https://doi.org/10.1147/sj.164.0363.

[90] Masaya Nakayama, Masaru Kitsuregawa and Mikio Takagi. “Hash-Partitioned Join Method Using Dynamic Destaging Strategy”. In: *Proceedings of VLDB 14*. VLDB ’88. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 468–478. isbn: 0934613753.

[91] Shiva Jahangiri, Michael J. Carey and Johann-Christoph Freytag. “Design Trade-Offs for a Robust Dynamic Hybrid Hash Join”. In: *Proc. VLDB Endow.* 15.10 (June 2022), pp. 2257–2269. issn: 2150-8097. url: https://doi.org/10.14778/3547305.3547327.

[92] Diane L. Davison and Goetz Graefe. “Memory-Contention Responsive Hash Joins”. In: *Proceedings of VLDB 20*. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 379–390. isbn: 1558601538.

[93] Antoine N. Mourad, Robert J.T. Morrarticle, Arun Swami and Honesty C. Young. “Limits of parallelism in hash join algorithms”. In: *Performance Evaluation* 20.1 (1994). Performance ’93, pp. 301–316. issn: 0166-5316. url: https://www.sciencedirect.com/science//pii/0166531694900191.

[94] Annita N. Wilschut, Jan Flokstra and Peter M. G. Apers. “Parallel evaluation of multi-join queries”. In: 24.2 (May 1995), pp. 115–126. issn: 0163-5808. url: https://doi.org/10.1145/568271.223803.

[95] Jens Teubner, Gustavo Alonso, Cagri Balkesen and M. Tamer Ozsu. “Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware”. In: *Proceedings of ICDE 2013*. ICDE ’13. USA: IEEE Computer Society, 2013, pp. 362–373. isbn: 9781467349093. url: https://doi.org/10.1109/ICDE.2013.6544839.

[96] Tim Gubner, Viktor Leis and Peter Boncz. “Optimistically Compressed Hash Tables & Strings in the USSR”. In: *SIGMOD Rec.* 50.1 (June 2021), pp. 60–67. issn: 0163-5808. url: https://doi.org/10.1145/3471485.3471500.

[97] Adam J. Storm, Christian Garcia-Arellano, Sam S. Lightstone, Yixin Diao and M. Surendra. “Adaptive self-tuning memory in DB2”. In: *Proceedings of VLDB 32*. VLDB ’06. Seoul, Korea: VLDB Endowment, 2006, pp. 1081–1092.

[98] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann and Alfons Kemper. “Massively Parallel NUMA-Aware Hash Joins”. In: *In Memory Data Management and Analysis*. Cham: Springer International Publishing, 2015, pp. 3–14. isbn: 978-3-319-13960-9.

[99] Yann Collet. *Zstandard - Fast real-time compression algorithm*. Facebook. 2015. url: https://github.com/facebook/zstd (visited on 15/01/2026).

[100] Maximilian Kuschewski, Jana Giceva, Thomas Neumann and Viktor Leis. “High-Performance Query Processing with NVMe Arrays: Spilling without Killing Performance”. In: *Proc. ACM Manag. Data* 2.6 (Dec. 2024). url: https://doi.org/10.1145/3698813.

[101] Amazon Web Services. *Amazon Elastic Block Store*. Amazon. 2021. url: https://aws.amazon.com/ebs/ (visited on 22/01/2024).

[102] Thomas Neumann and Alfons Kemper. “Unnesting Arbitrary Queries”. In: *Datenbanksysteme für Business, Technologie und Web*. Vol. P-241. LNI. Hamburg, Germany: GI, 2015, pp. 383–402. url: https://dl.gi.de/handle/20.500.12116/2418.

[103] Peter Boncz, Thomas Neumann and Viktor Leis. "FSST: fast random access string compression". In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2649–2661. issn: 2150-8097. url: https://doi.org/10.14778/3407790.3407851.

[104] Riki Otaki, Jun Hyuk Chang, Charles Benello, Aaron J. Elmore and Goetz Graefe. "Resource-Adaptive Query Execution with Paged Memory Management". In: *Proceedings of CIDR 2025*. Amsterdam, Netherlands: www.cidrdb.org, 2025. url: https://vldb.org/cidrdb/papers/2025/p2-otaki.pdf.

[105] Laurens Kuiper. *New Sorting Implementation*. DuckDB. 2025. url: https://github.com/duckdb/duckdb/pull/17584 (visited on 15/07/2025).

[106] Mark Raasveldt. *Support Parallel Order-Preserving Result Set Materialization*. DuckDB. 2022. url: https://github.com/duckdb/duckdb/pull/3700 (visited on 31/03/2025).