Technische Universität München

TUM School of Computation, Information and Technology

High-Performance Query Processing in the Cloud

Dominik Durner

Technische Universität München

TUM School of Computation, Information and Technology

High-Performance Query Processing in the Cloud

Dominik Durner

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitz:
Prof. Dr. Alfons Kemper

Prüfende der Dissertation:
1. Prof. Dr. Thomas Neumann
2. Prof. Anastasia Ailamaki
3. Prof. Dr. Viktor Leis

Die Dissertation wurde am 13.11.2023 bei der Technischen Universität München eingereicht und durch die TUM School of Computation, Information and Technology am 22.03.2024 angenommen.

Abstract

Independent scaling of compute and storage, a fundamental advantage of cloud infrastructure, presents unique opportunities for analytical query processing. The ever-increasing volume of data can now be stored economically with independently scalable cloud object storage. Cloud-based analytical database systems employ a disaggregated storage architecture to ensure the durability of data while also reducing cost by dedicating computational resources exclusively for query processing. Their elastic compute layer accesses structured, semi-structured, and unstructured data residing in cloud object storage. To achieve high-performance query processing and reduce cost in the cloud, the storage-related components of database systems require a redesign that reflects the architectural paradigm shift.

This thesis presents solutions to the distinct architectural challenges for analytics in cloud environments and achieves improvements for processing semi-structured data. First, our experimental study of cloud object storage guides the development of a cost-effective and highly performant storage engine. We present a blueprint for integrating cloud object storage retrieval into database systems. This integration leverages our novel retrieval library, which significantly reduces CPU consumption during object downloads. Second, commercially available cloud-native database systems use block-level caches to improve performance. When only a few data items are relevant within a block, these caches lead to poor storage space utilization. To address this issue, we advocate a new smart caching strategy that uses the access patterns of the workload to selectively store and process only dynamic regions of tables. Third, much of today's data originates from semi-structured formats, such as JSON, because developers prefer flexibility over upfront schema design. To accelerate semi-structured data analytics, we present a novel approach that materializes JSON data. By extracting and reordering blocks of JSON documents, our materialization exhibits performance comparable to columnar storage while being robust to heterogeneous datasets and facilitating query optimization. Finally, the growth of data introduces additional challenges for allocating memory during query processing. We investigate the impact of various memory allocators on analytical query performance.

Zusammenfassung

Die unabhängige Skalierbarkeit von Rechenleistung und Speicher ist eine grundlegende Neuerung der Cloud-Infrastruktur und bietet einzigartige Möglichkeiten für die analytische Datenverarbeitung. Die stetig wachsenden Datenmengen können nun mit unabhängig skalierbarem Cloud-Objektspeicher kostengünstig gespeichert werden. Analytische Cloud-Datenbanksysteme nutzen eine verteilte Speicherarchitektur, um die Dauerhaftigkeit der Daten zu gewährleisten und gleichzeitig die Kosten zu senken. Diese Kostenreduktion wird dadurch ermöglicht, dass die elastische Serverschicht nur während der Anfragebearbeitung für den Zugriff auf Daten im Cloud-Objektspeicher benötigt wird. Im Cloud-Objektspeicher liegen verschiedene Datentypen vor, die entweder strukturierte, semi-strukturierte oder unstrukturierte Daten enthalten. Um die Effizienz der Anfragebearbeitung zu erhöhen und die Kosten zu senken, müssen die speicherbezogenen Komponenten des Datenbanksystems an die Architekturänderungen in der Cloud angepasst werden.

In dieser Dissertation werden Lösungen für verschiedene architektonische Herausforderungen bei der Verarbeitung analytischer Anfragen in der Cloud entwickelt und Verbesserungen für die Analyse semi-strukturierter Daten erzielt. Zunächst dient unsere experimentelle Studie über Cloud-Objektspeicher als Grundlage für die Entwicklung einer kostengünstigen und hochperformanten Datenbanklösung für die Verarbeitung analytischer Anfragen. Wir verbessern das gleichzeitige Herunterladen und Analysieren von Daten aus Cloud-Objektspeichern durch die Integration unserer neu entwickelten Download-Bibliothek in Datenbanksysteme. Diese reduziert die Prozessorlast beim Herunterladen signifikant. Außerdem verwenden kommerziell erhältliche Cloud-Datenbanksysteme blockbasierte Zwischenspeicher, um die Performance zu verbessern. Wenn nur eine kleine Anzahl von Tupeln innerhalb eines Blocks relevant ist, können solche Zwischenspeicher zu einer schlechten Nutzung des lokalen Speicherplatzes führen. Um dieses Problem zu lösen, schlagen wir eine neue intelligente Zwischenspeicherstrategie vor, die die Semantik der Anfragehistorie nutzt, um selektiv einzelne Segmente von Tabellen zu speichern. Um die wachsende Menge semi-strukturierter Daten besser verarbeiten zu können, präsentieren wir zudem eine neue Strategie zur Analyse von JSON-Daten. Unsere Strategie extrahiert Daten aus JSON-Dokumenten mit gemeinsamen Schlüsseln als Blöcke und ordnet Dokumente für eine verbesserte Extraktion um, sodass eine schnelle Analyse möglich ist, ohne die Flexibilität des Formats zu beeinträchtigen. Zusätzlich entstehen mit zunehmender Datenmenge neue Herausforderungen für die Allokation von Arbeitsspeicher. Wir untersuchen daher, wie sich verschiedene Ansätze auf die Performance von analytischen Datenbanksystemen auswirken.

Acknowledgments

First of all, I would like to thank the members of my thesis committee, Prof. Thomas Neumann, Prof. Anastasia Ailamaki, Prof. Viktor Leis, and Prof. Alfons Kemper. Further, I extend my deep gratitude to all my co-authors for their exceptional work.

I am particularly thankful to Prof. Thomas Neumann for his guidance as my thesis advisor and for giving me the freedom to work on exciting and challenging topics of my choosing. I am grateful for the opportunity to be a part of your research group and for all the invaluable insights and inspirations during my doctorate.

A special thank you goes to my mentor, Prof. Viktor Leis, for always being available for advice and support. I would like to express my deep appreciation for helping me become a better researcher and collaborating on several publications.

As a member of the TUM database group, I had the opportunity to work with Prof. Alfons Kemper and Prof. Jana Giceva. Thank you for your helpful comments and the fruitful discussions.

During my PhD, I had the privilege of interning at Microsoft Research. I am grateful to my hosts, Badrish Chandramouli and Yinan Li, for this interesting experience and their collaboration on publishing the results of my internship.

I also want to thank my colleagues at TUM, Adrian Riedl, Adrian Vogelsgesang, Alex Khatskevich, Alexander Beischl, Alexander van Renen, Alexis Engelke, Alice Rey, Altan Birler, André Kohn, Andreas Kipf, Bernhard Radke, Christian Winter, Christoph Anneser, Ferdinand Gruber, Gabriel Haas, Harald Lang, Jan Böttcher, Linnea Passing, Lukas Vogel, Maximilian Bandle, Maximilian Kuschewski, Maximilian Reif, Maximilian Rieger, Maximilian Schüle, Michael Freitag, Michael Jungmair, Michail Georgoulakis Misegiannis, Moritz Sichert, Per Fuchs, Philipp Fent, Simon Ellmann, Stefan Lehner, Theresa Schmidkonz, Timo Kersten, Tobias Götz, Tobias Schmidt, and Varun Pandey, for the interesting discussions and helpful comments about my research and all our fun activities.

Finally, I would like to thank my friends, my family, and my partner. Without their continuous support and love, this thesis would not have been possible. A big thank you to my parents, Andrea and Gerhard, and my younger sister, Verena, for supporting me throughout my life. Last but not least, I would like to thank my partner, Christina, for being in my life and for your loving support. Without you, I would not have been able to complete this journey.

Funding. This project has received funding in part from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement number 725286).

Preface

This cumulative dissertation is based on the following peer-reviewed core publications. The published versions of the core publications are included in the appendix of this document. Excerpts and results of these papers are used in this thesis without additional labeling.

P1 Dominik Durner, Viktor Leis, and Thomas Neumann. “Exploiting Cloud Object Storage for High-Performance Analytics”. In: *PVLDB* 16.11 (2023), pp. 2769–2782

P2 Dominik Durner, Badrish Chandramouli, and Yinan Li. “Crystal: A Unified Cache Storage System for Analytical Databases”. In: *PVLDB* 14.11 (2021), pp. 2432–2444

P3 Dominik Durner, Viktor Leis, and Thomas Neumann. “JSON Tiles: Fast Analytics on Semi-Structured Data”. In: *SIGMOD*. ACM, 2021, pp. 445–458

P4 Dominik Durner, Viktor Leis, and Thomas Neumann. “Experimental Study of Memory Allocation for High-Performance Query Processing”. In: *ADMS@VLDB*. 2019, pp. 1–9

In addition to these publications, the author of this thesis also authored the following manuscripts, which are not part of this thesis:

1. Dominik Durner, Lennart Espe, Jana Giceva, and Anja Gruenheid. "TracEx: Understanding and Analyzing Database Traces". In: *CIDR*. 2024

2. Dominik Durner and Thomas Neumann. "No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System". In: *ICDE*. IEEE, 2019, pp. 734–745

3. Dominik Durner, Viktor Leis, and Thomas Neumann. "On the Impact of Memory Allocation on High-Performance Query Processing". In: *DaMoN*. ACM, 2019, 21:1–21:3 [Short Paper of P4]

In this dissertation, I will use the first person plural to acknowledge the contributions of my collaborators, although I am the principal researcher and sole first author of the included publications.

Contents

CHAPTER 1

Introduction

Data is the new oil. More and more data is created every day, and forecasts show that the growth rate of data is still increasing [Mil19]. Recent trends exacerbate this phenomenon because much of the data is generated without a human in the loop, such as sensor data or data from logging infrastructure. Nevertheless, even human-generated data is growing; for example, tracking page visitors across a website to study visitor patterns is critical to identifying business opportunities. Regardless of the source, it is essential to analyze large amounts of data to understand every aspect of a business or the reliability of a system.

Analytics on a large body of data. All this data is collected in database management systems (DBMS). Typically, these systems are classified as either transactional database systems or data warehouses, also known as analytical database systems, according to the workload they optimize for. Transactional database systems store current business information, such as orders and customer information. These systems are optimized for low latency on short-running transactions. On the other hand, data warehouses are periodically updated with extract-transform-load (ETL) pipelines from multiple sources, including transactional database systems [Kar+17]. Data warehouses are used to perform long-running online analytical processing (OLAP) workloads that would overwhelm transactional systems [Mon23a]. Consequently, it is crucial for analytical database systems to store, scan, and process large amounts of data.

Independent resource scaling. Traditionally, the collected data is stored on multiple disks located on on-premises servers in redundant array of inexpensive disks (RAID) configurations [Ora02]. On-premises solutions deploy distributed data warehouses on general-purpose server clusters to increase storage capacity beyond RAID configurations. This horizontal scaling of servers does not only increase storage space but also compute capacity. However, most of the time, only subsets of the data are queried, so the additional computing power is not used efficiently. Scaling nodes is expensive because more compute resources are added but partially unused, which results in high one-time cost and increased power consumption. Thus, independent scaling of storage and compute is desirable to reduce hardware cost.

Database systems move to the cloud. Public cloud offerings address this need. Cloud object stores enable the independent scaling of data cost-effectively (e.g., ~23$/TiB per month) [Ama23d] and can store petabytes of data. Additionally, they provide very strong durability guarantees; for example, AWS S3 promises a durability of 11 9's per year [Ama23e]. Data warehouses are now taking advantage of this trend by adopting a tiered or disaggregated storage model. Their elastic compute tier accesses data that persists on independently scalable remote cloud object storage, e.g., Amazon S3, Google Cloud Storage, and Azure Blobs [Ama23b; Goo23; Mic23]. Today, almost all data warehouse systems, such as AWS *Redshift*, Google *Big Query*, Azure *Synapse*, Apache *Spark*, and Apache *Presto*, support querying cloud storage [AR20; Arm+15; Dag+16; Gup+15; Mel+10]. For improving latency and throughput, most systems rely on local caching during processing.

Main memory is a scarce resource. Disaggregating storage and computation simplifies storing large amounts of data (up to petabytes per customer) [Ama23c]. In contrast to persistent disk storage, main memory is not easily scalable. Over the last decade, the anticipated strong reduction in the cost of main memory and the increase in single server memory capacity have not occurred [NF20]. Although most queries only touch a subset of the data, some queries produce large intermediate query results that scale with the amount of the input data; for example, chokepoint query Q13 of the TPC-H workload [BNE13]. Careful management of this scarce resource is necessary to ensure sufficient main memory for every query.

Semi-structured data is common. Besides classical structured relations, many interfaces rely on semi-structured formats to exchange data. Semi-structured formats interleave schema and content of the data within the same document. Developers often favor these formats due to their ease of creation and transfer, as all relevant information to access the data is contained within each document. Notably, the JavaScript Object Notation (JSON) is commonly used by many web applications [Li+17]. Because of its widespread use for data transfer, large JSON datasets are accumulated by logging software events or collected from publicly facing APIs [Met23; X C23; Yel23]. Analytics on such datasets is valuable, but the interleaving of data and schema poses a challenge during data scanning. Accessing the desired keys requires a traversal of the document. Efficiently handling semi-structured data necessitates adaptations in the storage layer of database systems.

Opportunities for database systems. From these observations, this thesis identifies several challenges and opportunities for improving analytical query processing in the cloud. In the remainder of this thesis, specific research questions are devised prior to discussing novel solutions to these issues, which were

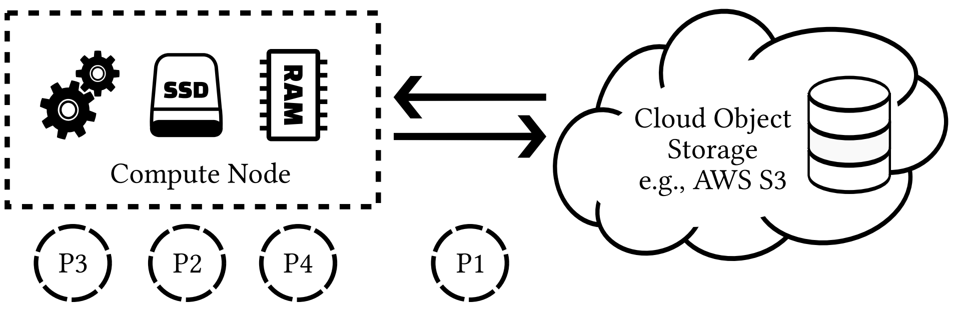


Figure 1.1: Our papers target different aspects of cloud-based data processing.

first presented in the publications included in this thesis. By leveraging our novel solutions, database systems become more cost-effective and achieve higher performance in the cloud. Figure 1.1 shows a schematic overview of the four publications in the architectural landscape of cloud-based data processing:

- Paper P1 demonstrates efficient, inexpensive, and CPU-conscious retrieval.
- Paper P2 explores better local caching by improving SSD space utilization.
- Paper P3 facilitates fast and robust processing of semi-structured data.
- Paper P4 analyzes the impact of memory allocation for query processing.

1.1 High-Performance and Cost-Effective Analytics on Cloud Object Storage

Network bandwidth increases fourfold. Until recently, the primary issue of using cloud object storage for analytical purposes stemmed from the low-bandwidth network connection between the compute tier and the storage infrastructure. Thus, the legacy database system design relies on storing large amounts of data on local SSDs (~2 GiB/s per dedicated SSD at AWS). In 2018, instances with 100 Gbit/s (≈ 12 GiB/s) Ethernet networking capabilities were introduced at AWS [Bar18; BL19]. This resulted in a fourfold increase in the bandwidth available per instance. Compared to clusters connected with Infiniband, 100 Gbit/s Ethernet is both cost-effective and readily available in public clouds.

Accessing data from cloud object storage is cost-effective. To illustrate this cost-effectiveness, we compare the on-demand pricing for two related instance types on AWS: `c5n.18xlarge`, which offers 100 Gbit/s networking, and `c5.18xlarge`, which provides 25 Gbit/s networking [Ama23a]. Although the `c5n` instance provides a fourfold boost in network throughput and has more main memory than the `c5`, it only results in a 22% increase in cost. This networking boost closes the performance gap between remote network and local NVMe SSD bandwidth, making cloud storage a more compelling choice for workloads that heavily rely on bandwidth. For example, consider the `i3en.24xlarge` instance, known for its local NVMe bandwidth. This instance provides a read bandwidth of 16 GB/s, similar to the full-duplex 12 GB/s network bandwidth of the `c5n.18xlarge`.

Persistent data is stored on cloud object storage. Nowadays, most cloud-based database systems leverage cloud object storage for persisting data. Because caching was already used in on-premises environments, many different strategies were developed to avoid retrieving data from remote nodes [Jal+18; Yan+21; Zha+22]. Recent work on using serverless cloud functions for analytics [BSA23; MMA20; Per+20], such as *Starling* and *Lambada*, focuses on direct processing of data from remote storage, as serverless functions cannot utilize local state. These functions, however, have very limited bandwidth per invoked function compared to modern network-optimized instances (e.g., a factor of 100×). Although the high (first-byte) latency of remote storage has been discussed in previous work [BA22], no comprehensive empirical study of analytics on cloud object storage has been conducted examining both its performance (i.e., bandwidth, latency, and concurrency in multiple configurations) and cost characteristics.

Challenges for analytics using high-bandwidth networks. Efficient use of cloud object storage in database systems poses several key challenges. (i) To fully utilize high-bandwidth networks, a large number of concurrent requests must be outstanding due to the low bandwidth of individual object requests. This necessitates a careful retrieval integration into the database system to fully exploit the bandwidth potential. (ii) Retrieving data from the network (via TCP) causes higher CPU overhead than from local disks, which reduces the number of cores available for concurrent analytics. Since query engines compete for computational resources, the CPU cores spent on network retrieval must be minimized to ensure efficient data analysis. (iii) Many cloud database systems support running in different cloud environments, allowing users to choose their preferred cloud vendor. However, each cloud vendor typically offers its own cloud object storage library. Integrating multiple networking libraries to support multi-cloud functionality adds complexity to the system.

Exploiting the network bandwidth potential while optimizing cost. Efficient analytics on data residing in cloud object storage is achieved by overcoming the three mentioned challenges, taking full advantage of the instance's bandwidth while maximizing CPU resources for analytical query processing. In addition to the query performance, the cost of analytics is equally important in the cloud. Therefore, a cloud-native database system must optimize for both performance and cost characteristics, which leads to the following research question:

> ***Research Question 1:*** *Given the high network bandwidth and limited CPU resources on a single instance, how can database systems efficiently and cost-effectively analyze data from cloud object storage?*

Enabling cost-effective and high-performance analytics. Paper P1 of this dissertation addresses performance and cost challenges, offering an innovative approach for efficient analytics on data stored in disaggregated cloud object stores. To this end, we first conduct an experimental study of cloud object stores to optimize request size and concurrency for cost-effective and high-throughput retrieval. The results of the experiments are used to develop *AnyBlob*, a low-overhead multi-cloud download manager that reduces CPU resource consumption while maintaining instance throughput. Finally, we demonstrate a blueprint for seamlessly integrating *AnyBlob* into database engines, enabling efficient analytics by interleaving object retrieval with analytical processing. Our integration facilitates inexpensive and efficient analytics at instance bandwidth on data stored in cloud object storage, evaluated by incorporating *AnyBlob* into our database system *Umbra* [NF20]. *Umbra* is a full-fledged database system that generates executable code for queries [Gru+23; KLN21; KLN18; Neu11] and introduces many high-performance analytical operators [BGN21; FN21; Fre+20; NLK17; SN22; Win+22; Win+20].

1.2 Caching for Database Systems in Disaggregated Storage Environments

Renewed interest in local caching. The transition from shared-nothing database systems to the storage-disaggregated architecture in the cloud has also sparked renewed interest in local node data caching. The majority of cloud database systems cache data on local compute node disks to reduce access latency or increase query throughput. Although high-bandwidth networks are becoming more affordable, not all cloud vendors provide 100 Gbit/s networking for general-purpose instances. Even in the presence of high-bandwidth networking, the additional utilization of local SSDs can improve the overall data bandwidth, resulting in better query performance. Therefore, cloud database systems aim to store hot data on the compute layer's fast local storage, e.g., SSDs. But, the limited capacity of these fast storages necessitates a sophisticated caching strategy. Caches, such as the *Alluxio* analytics accelerator [All23; Li18;

Li+14], the Databricks *Delta Cache* [Arm+20; Dat23a], and the *Snowflake Cache Layer* [Dag+16], are widely used in commercial cloud offerings.

Low cache utilization in block-level solutions. These caching solutions usually operate as black-box systems. They cache at the file or block level and employ standard cache replacement policies like LRU for cache management. Block-level caching is commonly used because most data is already stored in partition attributes across (PAX) layout [Ail+01], e.g., Apache Parquet [Apa23d; Zen+23]. However, the existing solutions often suffer from low storage space utilization, as a single record of interest results in the caching of the entire block, wasting precious storage space. This problem persists even with optimized columnar formats that employ zone maps to skip irrelevant blocks [Gra09; Ora17; Sun+14]. Additionally, cloud databases increasingly support analytics on heterogeneous data such as CSV, JSON, and row-oriented binary formats like Apache Avro [Apa23c]. Because these formats do not follow a columnar layout, block-level caching is often insufficient for analytics. Although LRU-based caches use the history for deciding which blocks to keep, they fail to capture the semantics of the workload, e.g., relevant table filters.

Caching filtered data tailored to the workload. Because current solutions fail to capture workload semantics and operate at block granularity, local node caches often store irrelevant data. Due to the limited storage space on local nodes, it is important to utilize the caches as efficiently as possible. Storing filtered data that is tailored to the workload promises to increase this storage space utilization and may lead to better performance, which is summarized in the following research question:

> ***Research Question 2:*** *What performance impact does smart caching of filtered data tailored to the observed workload have on analytics, considering disaggregated storage with limited local storage?*

Crystal transparently manages a semantic cache. To overcome the low utilization of caches, Paper P2 of this thesis proposes *Crystal* – a smart storage middleware. Positioned between the database system and raw storage, *Crystal* operates as a cache management system (CMS) dedicated to storage tasks on the compute node. It efficiently manages high-speed local storage as a cache and retrieves data from remote storage with the help of a download library, such as the cloud-vendor-provided libraries or *AnyBlob*. Unlike traditional block-level caches, *Crystal* dynamically selects which data regions (i.e., table rows) to transform and to cache locally in a columnar format based on the observed workload. If helpful for query performance, data can be cached in multiple regions. Lightweight connectors facilitate the communication between

Crystal and the host database system. First, the database system submits data requests with filters on the relation. *Crystal* then returns the file path of the requested data, which may reside locally in the cache or remotely on cloud object storage. In essence, *Crystal* acts as a bridge between storage and database systems. It enables efficient caching into semantically meaningful regions in a DBMS-agnostic columnar format.

1.3 Fast Analytics on Semi-Structured Data

DBMSs must efficiently process JSON. As more and more semi-structured data is generated, database systems must strive to store and analyze such modern formats. Due to its widespread use in web applications, semi-structured data is often transferred as human-readable JSON. Because of the importance of analyzing these formats, several methods have been proposed to improve the performance of JSON analysis.

Different strategies for handling JSON documents. Relational database systems use three main strategies when dealing with JSON documents: They either scan the raw files as an external table, store each object as a string, or use an optimized binary representation on a per-object basis [Pos23b]. Fast processing of raw data in database systems has been explored for both structured [Idr+11] and semi-structured formats, coupled with code generation and indexes for faster subsequent access [KAA16]. *SIMD-JSON* [LL19] and *Mison* [Li+17] allow JSON parsing at rates up to one GiB per second per core. However, querying raw JSON documents remains expensive because the entire dataset must be parsed to access a single field. Storing complete documents, either as plain text or binary-optimized, in the database requires subsequent accesses into the documents when analyzing them. Accessing keys within documents has high performance overhead compared to reading values from a columnar storage, commonly used in relational databases. *Sinew* [TDA14] has been developed to improve data access by extracting entire value columns from JSON documents. This extraction is based on the observation that documents from the same system have a similar structural schema. However, the effectiveness of *Sinew* depends on a globally inherent internal document structure. Due to its global structure analysis, dynamic or heterogeneous documents pose robustness challenges, and the update process is resource-intensive. *Dremel* [Mel+10], implemented in Apache Parquet [Apa23d], splits documents at the record level (shredding) and reassembles them during query execution. However, this reassembly requires an access automaton that becomes expensive during analysis. Processing Parquet files is, therefore, often CPU-bound. This even applies to relational files without nested and optional components [Can17].

Robustly exploiting the common structure of documents. To analyze JSON data efficiently, modern database systems must strive to achieve high access performance, maintain robustness regarding heterogeneous data, and facilitate join ordering. Because most documents are machine-generated, they have a fairly predictable structure, which can be used for materializing columns. Although these columns provide high access performance, documents may change over time, or different document types may be inserted, making a global extraction unusable. The following research question explores the utilization of the inherent document schema for robust materialization:

> ***Research Question 3:*** *How can database systems efficiently infer and exploit the structural information of a set of documents to enable fast and robust analytics on semi-structured data?*

JSON tiles enables fast analytical query processing. To improve JSON processing in database systems, Paper P3 of this dissertation presents *JSON tiles*, a comprehensive set of algorithms and methods to facilitate high-performance analysis. *JSON tiles* automatically identifies the underlying common structure within a group of documents, called a tile. We use this structural insight to infer data types, instantiate common keys as relational columns, and generate statistics for query optimization. With these strategies, *JSON tiles* achieves analytical query performance on JSON data comparable to a native relational columnar storage. For effectively handling heterogeneous data, *JSON tiles* reorders documents between tiles to increase the number of extractable column chunks. If uncommon data is inserted, we use an optimized binary format, enabling fast key lookups. These techniques are automatic and transparent, so *JSON tiles* does not sacrifice the flexibility of JSON documents.

1.4 Impact of Memory Allocation on Analytical Query Performance

Workload dictates the memory allocation pattern. With cloud storage being virtually unlimited, the amount of stored data increases faster than the size of main memory. In analytical workloads, processing large datasets requires the allocation of a significant amount of memory. Due to advancements in the query performance of modern database systems, components are becoming performance bottlenecks that were not a concern in traditional database systems, such as memory management. Memory allocations and deallocations notably impact query performance, evident in performance profiling. As the allocation pattern depends on the workload, efficiently managing large amounts of memory operations is crucial.

Modern hardware stresses allocation bottlenecks. Modern hardware amplifies this issue, as hundreds of general-purpose cores perform memory operations simultaneously. Moreover, modern servers employ a many-core architecture that consists of multiple CPU nodes. Accessing data from a remote non-uniform memory access (NUMA) node is more expensive than accessing local memory. Since memory allocators implement various strategies for allocating and deallocating memory, the choice of the memory allocator affects query processing. We classify dynamic memory allocators based on four fundamental properties:

- **Performance.** How much CPU time is spent for malloc and free?
- **Scalability.** How well does the allocator scale on larger machines, and how much overhead is introduced for multi-threaded allocations?
- **Memory Fairness.** Is the freed memory returned to the operating system so that other processes can reuse it?
- **Memory Efficiency.** How much of the memory is unusable due to fragmentation?

Understanding the impact of memory allocator choice. Despite being a critical factor in query processing [App+17], no prior empirical study has been conducted on different dynamic memory allocation approaches for analytical database systems. As memory allocators vary significantly in their allocation strategy, it is important to understand the implications of selecting a particular memory allocator, which is the focus of the following research question:

> ***Research Question 4:*** *What impact does the selection of the memory allocator have on query performance, scalability, and memory efficiency of analytical database systems?*

Memory allocator analysis reveals performance differences. Paper P4 of this thesis presents a thorough analysis of the impact of memory allocation on high-performance query processing. We evaluate various approaches on analytical workloads according to the four dynamic memory allocator characteristics. Our study reveals that the memory allocation strategy significantly impacts query performance.

CHAPTER 2

Research Methodology

Methodology outline. This chapter discusses the details of the research methodologies used in this publication-based dissertation. Prior to investigating the specifics of each paper, this chapter outlines the general research approach in the field of database systems.

The research process. Every research topic stems from identifying problems and reviewing related work. Unresolved issues lead to open research questions, driven by a desire to adapt to modern technological developments or achieve verifiable improvements to a particular system. From this research question, various ideas lead to testable research hypotheses that guide the research process. In the database systems community, these hypotheses must be tested against a modified system or prototype to validate whether they hold. This development is a multi-stage process. First, the ideas that guide to the hypotheses are used to create a system design. Then, this design is put into practice by implementing new components of a system, replacing existing components with newly developed ones, or implementing an entirely new prototype system. Finally, the newly developed components are evaluated against state-of-the-art approaches on standardized workloads that resemble real-world behavior. This process is typically iterative and incremental. Adaptations and new ideas arise from observations during the development and evaluation of test data. These ideas are incorporated into the system to enhance its observable performance. The steps are visualized in Figure 2.1.

Scientific contribution and section outline. In the following sections, a short summary of the challenges introduces each paper's research question. One or more verifiable hypotheses are formulated for each research question. Subsequently, this thesis highlights the key scientific contributions of our publications. A description of the testbed and a brief evaluation will be used to verify the stated hypotheses. More information about the scientific contribution and the experimental evaluation can be found in the original papers included in this thesis. Please also refer to these publications for detailed explanations and discussions.

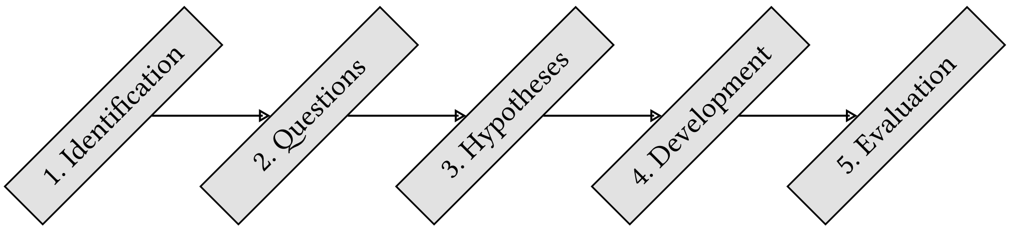


Figure 2.1: The five steps of the research process.

2.1 Paper P1. Exploiting Cloud Object Storage for High-Performance Analytics

Exploiting cloud object storage for analytics. The emergence of inexpensive network technologies that support 100+ Gbit/s makes retrieving large amounts of data from external storage more attractive. Despite the widespread use of cloud object storage, it is unclear how to efficiently retrieve data from cloud object storage while simultaneously processing queries. Because a single request only has limited bandwidth, many requests need to be outstanding simultaneously to saturate the instance bandwidth. This large number of requests makes the design and integration of cloud object storage retrieval challenging. Moreover, the CPU resource utilization for data retrieval from the network is significantly higher than for local storage drives. These observations lead to the following research question:

> ***Research Question 1.*** *Given the high network bandwidth and limited CPU resources on a single instance, how can database systems efficiently and cost-effectively analyze data from cloud object storage?*

2.1.1 Hypotheses

Reducing CPU overhead of retrieval is crucial. HTTP is the most common interface to cloud object storage. Unfortunately, this connection requires more CPU resources than retrieving data from locally attached disks. Retrieving data should use as few CPU cores as possible to free resources for query processing. Thus, high-performance query processing relies on resource-conscious retrieval. Modern IO stacks in the Linux kernel promise lower CPU usage. In particular, io_uring communicates with the kernel via inexpensive shared ring buffers [Axb19]. We anticipate lower CPU resource usage while retaining the same throughput, as stated by the following hypothesis:

> ***Hypothesis 1.1:*** *By using io_uring, a modern kernel stack for IO, and an optimized download process, the CPU utilization for retrieving data from cloud object storage can be significantly reduced without affecting download performance.*

Queries on remote data are processed at network line rate. Due to the high bandwidth of network-optimized instances, processing at network line rate could provide an end-to-end bandwidth comparable to local SSDs in RAID configurations. Our system, even with disabled caches, should achieve query throughput for bandwidth-dominated workloads similar to commercially available cloud analytics systems that utilize local disks. Therefore, the following hypothesis should be experimentally validated:

> ***Hypothesis 1.2:*** *Database systems can efficiently retrieve and analyze data from cloud object storage with a bandwidth utilization that is close to the network bandwidth limit of the instance.*

Analytics on cloud object storage is cost-effective. Besides performance, cost is the major decision driver in cloud environments. Storing data on cloud object storage is inexpensive, but additional access cost occur [Ama23d]. By optimizing the request size, the storage access cost can be bound by the compute instance cost, enabling cost-conscious processing that avoids unexpectedly high bills. In accordance with cost simulations [LK21], processing queries in public clouds should be multiple times cheaper in comparison to what users pay for commercially available solutions. The following testable hypothesis summarizes our expected results:

> ***Hypothesis 1.3:*** *Since careful analytics on cloud object storage is cost effective, database systems can considerably reduce analytics cost compared to commercial offerings while providing comparable throughput.*

2.1.2 Scientific Method and Design

Characteristics of cloud object storage guide the design. To validate the hypotheses, our goal is to integrate high-performance and cost-effective query processing from remote storage in a modern relational database system. Our contribution is three-fold. (i) To obtain a deeper understanding, we conduct an extensive experimental study on the characteristics of cloud object storage. The results of this study guide our development in achieving high-performance and inexpensive analytics. (ii) Preliminary results indicate that retrieving data from remote storage is particularly demanding on the CPU. In order to decrease the

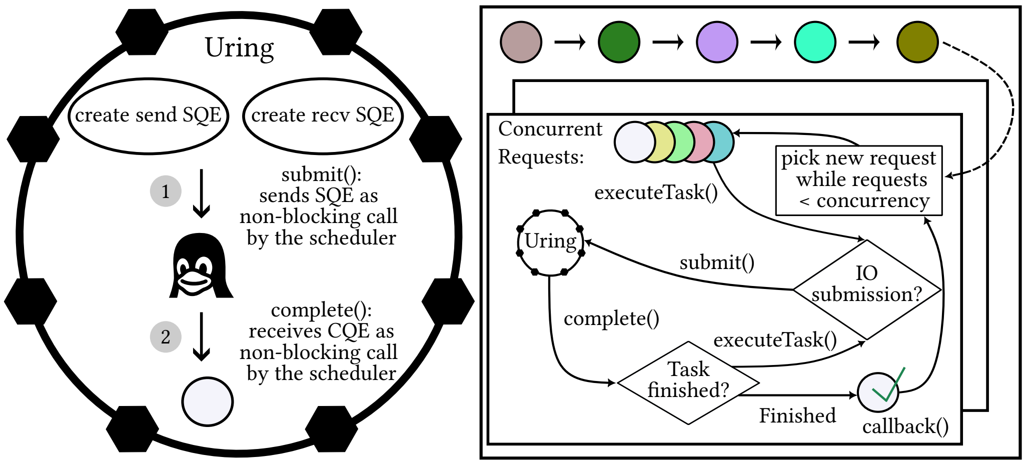

Uring
create send SQE
create recv SQE
1
submit(): sends SQE as non-blocking call by the scheduler
2
complete(): receives CQE as non-blocking call by the scheduler
Concurrent Requests:
pick new request while requests < concurrency
executeTask()
Uring
submit()
IO submission?
complete()
executeTask()
Task finished?
Finished
callback()

Figure 2.2: AnyBlob uses io_uring to asynchronously process state-machine-based message tasks. Concurrency within a single thread facilitates higher performance without thread oversubscription.

number of cores required for retrieving data, we create *AnyBlob*, an innovative retrieval library. (iii) With *AnyBlob* and the results of the study, we demonstrate a seamless integration of cloud object storage retrieval into database systems.

Cloud storage allows for high-bandwidth retrieval. Our study examines the cost and durability service level agreements (SLAs) of various storage options, including AWS S3, EBS, and local disks. Cloud object storage provides the highest durability guarantees while also being the most cost-effective option. We conduct experiments on multiple cloud vendors, demonstrating the latency characteristics, total retrieval bandwidth, required concurrency of requests for achieving the total bandwidth, and cost implications of the request size. Interestingly, retrieving data from cloud object storage within a single location (e.g., an AWS region) incurs only fixed access cost independent of request size and monthly storage cost. Therefore, the larger the requests, the more data can be retrieved per dollar, universal across all major cloud vendors. However, database applications that rely on pruning unnecessary blocks prefer smaller requests. Considering that the retrieval throughput plateaus when using request sizes of several MiB, we recommend a size of $8 - 16$ MiB, which we deem cost-throughput optimal. Although a single object request only has limited bandwidth (i.e., high latency per request), the full instance bandwidth of 100 Gbit/s can be exploited by retrieving hundreds of requests simultaneously.

More CPU resources for analytics due to AnyBlob. We use these findings to create a low-overhead, open-source, and multi-cloud retrieval manager, called *AnyBlob* [Dur22]. *AnyBlob*'s objective is to achieve instance network bandwidth

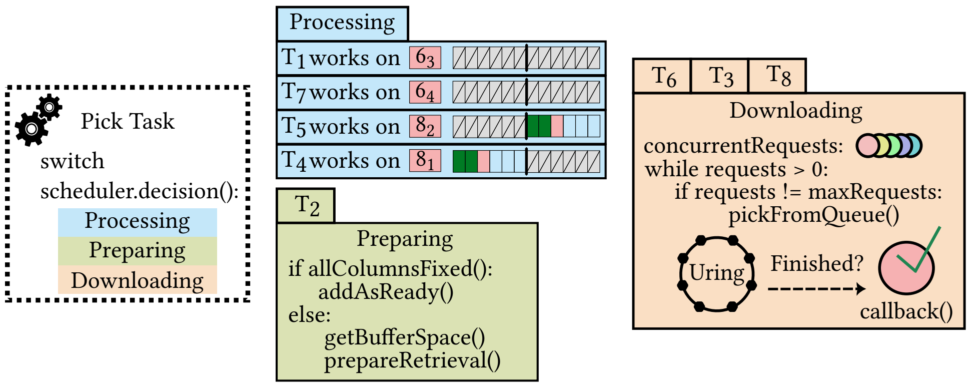


Figure 2.3: Table scan design using different worker tasks. In this example, four threads process queries, one prepares new requests, and three download data.

comparable to cloud-vendor-provided libraries while reducing CPU usage. To avoid performance penalties and ensure smooth integration into the database system's worker thread model, *AnyBlob* must avoid thread oversubscription, such that we do not use more download threads than hardware threads. *AnyBlob* uses asynchronous networking based on the modern Linux kernel interface io_uring, allowing a single thread to handle multiple concurrent requests. It maintains a state machine for each request and suspends the request after generating a new io_uring-based syscall. Once the syscall is completed, the task is reevaluated, and new syscalls are scheduled if more network communication is required or a callback finishes the task. While a single-threaded approach that handles multiple concurrent requests already has high network throughput, it is not sufficient for saturating modern high-bandwidth instances. Thus, multiple retriever threads are grouped for better performance while retaining a simple user interface via a group-global request queue. The design of *AnyBlob* with two retriever threads that share one global queue is depicted in Figure 2.2.

Seamless object retrieval integration. To integrate data retrieval from cloud object storage into database systems, we utilize the findings of our experimental study and *AnyBlob* for resource-conscious and high-performance object retrieval. Our integration, shown in Figure 2.3, takes advantage of the task-based design of database systems and the internal management of database worker threads. All major database systems require and implement the concept of tasks to switch execution between different queries (e.g., in multi-tenant systems), preventing a single long-running query from stalling the DBMS. We can repurpose worker threads in this task system to retrieve data. At task boundaries, the job of a worker thread can be adjusted, facilitating high adaptivity. In our design, a novel scheduler determines the current job of a worker thread during table scanning. It balances retrieval and processing performance by using observed throughput statistics. The worker threads are extended to also support data preparation and data retrieval. During unmodified worker tasks, the scan data is simply processed and passed to the upper-level operators. New requests are generated during a preparation task, and memory in the buffer manager is allocated for downloading. The download task schedules the worker as an *AnyBlob* retrieval thread that continuously downloads the objects of the currently processed table.

2.1.3 Implementation Overview

AnyBlob is a multi-cloud and resource-conscious object retrieval library. *AnyBlob* is designed as a library to be used in other programs. It is written in C++ and relies on modern Linux kernel features. At its core, it processes requests and retrieves or sends data to cloud object storage. The network communication is handled via io_uring's shared buffers. Although all cloud object stores communicate via HTTP messages, the messages themselves differ between vendors. We implement interfaces to multiple vendors, and our design allows for an easy integration of more vendors. Authentication and encryption are integrated into *AnyBlob* with the help of the OpenSSL library [Ope23].

Buffer manager and storage engine adaptations. We implement a new storage engine within our modern database system *Umbra*, written in C++, that enables reading from columnar chunks (PAX layout). Our format is based on data blocks with small materialized aggregates (SMAs) [Lan+16; Moe98]. We retrieve objects without additional userspace data copies by downloading them directly into pages of our buffer manager. In contrast to regular pages of the buffer manager, pages for remote retrieval are discarded at eviction and are not written back to disk. Our buffer manager transparently manages both page types, unifying the used buffer space.

2.1.4 Experimental Evaluation and Conclusion

Testbed. The experiments are conducted on AWS within the `eu-central-1` region. Our data and metadata structure, stored on AWS S3, resembles Apache *Iceberg* and *Delta Lake* [Apa23b; Dat23b; Hug21]. Except for *Snowflake*, all measurements are executed using a single instance of `c5n.18xlarge`, comprising 72 vCPUs (36 cores, 72 threads), 192 GiB of RAM, and 100 Gbit/s Ethernet. For *Snowflake*, we provision a large data warehouse. All experiments that compare to *AnyBlob* leverage version 1.9.140 of the AWS C++ SDK [Ama23g].

Reducing CPU usage of retrieval. Hypothesis 1.1 states that *AnyBlob* reduces CPU utilization without impacting performance. Figure 2.4 illustrates three different retrieval strategies: *AnyBlob*, AWS *SDK* [Ama23g], and AWS *SDK*

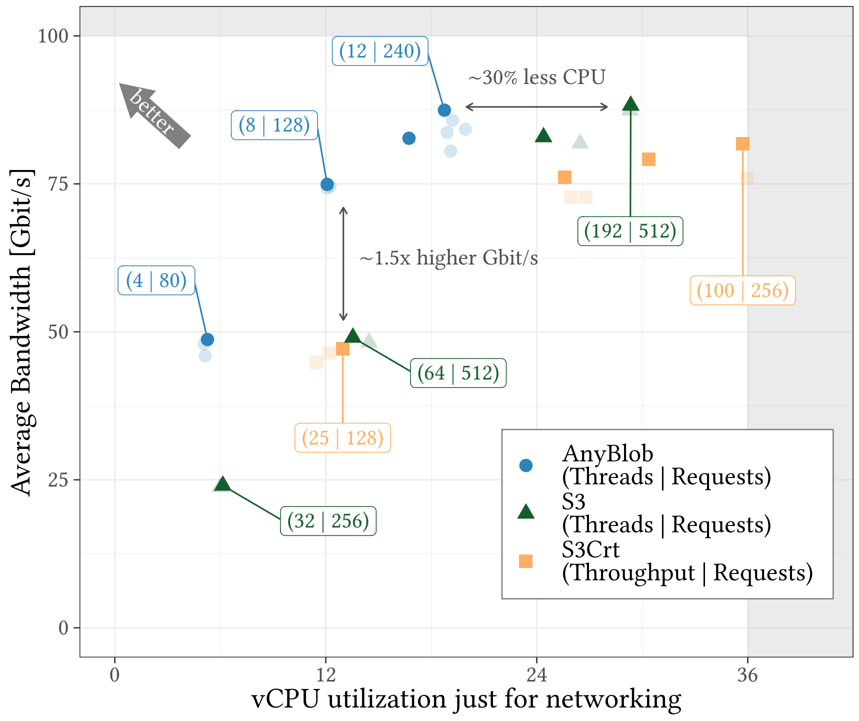


Figure 2.4: Throughput and CPU usage Pareto curves for AnyBlob, AWS C++ SDK S3, and AWS C++ SDK S3Crt libraries.

Crt [Ama23f]. The performance and CPU utilization Pareto curve demonstrates that *AnyBlob* outperforms all other approaches. *AnyBlob* achieves the same maximum throughput while decreasing CPU utilization by 30%. This experiment validates Hypothesis 1.1 since our approach dominates all points of the Pareto curve.

Achieving instance bandwidth during analytics. Hypothesis 1.2 asserts that *AnyBlob*-enabled *Umbra* can saturate the instance network bandwidth during analytical query processing. To showcase this, we compare a remote-only version of *Umbra* with disabled caches to an in-memory *Umbra* on the same instance. The remote *Umbra* version discards previously downloaded blocks and retrieves them again. Table 2.1 presents different chokepoint TPC-H queries for both versions [BNE13; Dre+20]. We distinguish between queries that are limited by available bandwidth and those that are limited by computation. For queries that are limited by bandwidth, specifically Q1, Q6, and Q19, we observe end-to-end bandwidths of 75 Gbit/s or more, which is close to the instance limit. Note that this bandwidth is only a lower bound because it is calculated based on the total amount of data retrieved and the end-to-end query latency. The

Table 2.1: In-memory and remote-only Umbra performance and cost comparison (TPC-H, SF 500).

Query	Q1	Q3	Q6	Q9	Q13	Q18	Q19	GM
In-Memory [s]	1.14	2.93	0.52	10.61	9.50	18.91	0.74	2.03
Remote [s]	3.52	5.87	2.47	13.34	12.47	22.20	3.82	4.94
Factor [×]	3.08	2.01	4.78	1.26	1.31	1.17	5.15	2.42
Bandwidth [Gbit/s]	75.00	55.76	77.73	40.67	30.86	15.41	76.87	49.80
Cost S3 [¢]	0.29	0.21	0.17	0.31	0.28	0.22	0.25	0.15
Cost EC2 (on demand) [¢]	0.38	0.63	0.27	1.44	1.34	2.39	0.41	0.53

observed bandwidth is comparable to the combined throughput of 5 NVMe SSDs in the cloud (~2 GiB/s at AWS). Computation-bound queries, for example Q9, Q13, and Q18, have little performance difference between the fully remote and in-memory databases. In these scenarios, our retrieval process has only negligible CPU overhead. In summary, this experiment confirms Hypothesis 1.2.

Cost-effective analytics from cloud object storage. With the shift to cloud computing, cost is now as important as performance. Hypothesis 1.3 claims that cloud object storage analytics can be cost-effectively implemented, and that database systems can process data much cheaper than current cloud offerings. To verify this claim, we compare our remote-only *Umbra* version against a self-hosted Apache *Spark* and a *Snowflake* warehouse of size L on TPC-H queries. Table 2.2 shows that using *Umbra* on a spot instance is 6× less expensive than using the configured *Snowflake* warehouse. Furthermore, *Umbra* is significantly more affordable than a cold *Snowflake* warehouse, which we shut down after each query to flush the cache. Note that this *Umbra* version actively discards previously downloaded data and redownloads the necessary information, similar to a cold *Snowflake* warehouse. Despite the huge cost reduction, *Umbra*'s performance is similar to the best competitor. We also compared costs with *Spark* and *Athena*[1] but found that these systems are ten times more expensive than our proposed solution. This significant cost difference verifies Hypothesis 1.3.

AnyBlob enables fast cloud analytics in DBMS. In summary, the study of the characteristics of cloud object storage drives our development of *AnyBlob*. *AnyBlob* allows for a significantly reduced CPU consumption while achieving instance bandwidth. The low number of threads required for downloading simplifies the integration of retrieval into analytical database systems. The experiments show that *Umbra* with *AnyBlob* is a cost-effective and high-performing analytical query engine, even if all data requires remote retrieval.

[1]The Athena cost use a back-of-the-envelope calculation given a columnar format and block-level pruning similar to our data block design. No caching is considered.

Table 2.2: Performance and cost comparison on all TPC-H (SF 1000) queries. For Umbra (remote-only version), we show costs for running the queries with spot instances (may be terminated) and on-demand instances (most expensive).

	Umbra (c5n.18xlarge)			Snowflake Cached (L)		Snowflake Remote (L)		Spark (c5n.18xlarge)		Athena[1] (calc.)
	[s]	[$¢_S$]	[$¢_D$]	[s]	[¢]	[s]	[¢]	[s]	[$¢_S$]	[¢]
Q1	**6.82**	**0.82**	1.31	7.51	3.34	17.56	7.80	646.77	23.10	33.00
Q2	**3.84**	**0.22**	0.50	4.31	1.91	7.13	3.17	67.03	2.48	11.13
Q3	12.82	**0.87**	1.80	**11.45**	5.09	19.95	8.87	449.70	16.15	40.87
Q4	9.27	**0.64**	1.31	**7.05**	3.13	13.48	5.99	132.23	4.85	29.25
Q5	13.52	**0.88**	1.86	**12.38**	5.50	21.66	9.63	814.73	29.14	39.08
Q6	5.32	**0.52**	0.90	**2.84**	1.26	13.56	6.03	61.47	2.29	24.00
Q7	13.21	**0.94**	1.90	**10.63**	4.72	20.44	9.09	893.88	31.96	43.62
Q8	14.30	**1.01**	2.04	**11.93**	5.30	20.99	9.33	1001.26	35.78	53.46
Q9	30.69	**1.81**	4.03	**30.26**	13.45	38.65	17.18	1151.45	41.14	78.24
Q10	**15.73**	**1.16**	2.30	21.21	9.43	26.68	11.86	140.70	5.18	42.33
Q11	**1.85**	**0.11**	0.24	2.66	1.18	5.53	2.46	66.58	2.45	4.47
Q12	**9.24**	**0.78**	1.45	9.33	4.15	13.31	5.91	110.40	4.07	36.75
Q13	25.24	**1.46**	3.28	**18.79**	8.35	21.01	9.34	109.09	3.93	48.12
Q14	7.88	**0.62**	1.18	**6.65**	2.95	16.39	7.28	109.41	4.00	28.23
Q15	**8.01**	**0.62**	1.20	9.29	4.13	21.98	9.77	123.08	4.60	33.43
Q16	**4.72**	**0.21**	0.55	6.67	2.97	9.28	4.13	32.65	1.21	3.58
Q17	9.13	**0.75**	1.40	**7.70**	3.42	18.86	8.38	892.74	31.96	39.95
Q18	47.13	**2.12**	5.52	**29.05**	12.91	31.64	14.06	1322.92	47.31	42.75
Q19	**7.08**	**0.76**	1.27	8.90	3.96	18.94	8.42	121.27	4.42	30.82
Q20	9.15	**0.74**	1.41	**8.64**	3.84	21.78	9.68	127.79	4.71	41.79
Q21	24.29	**1.73**	3.49	**21.56**	9.58	30.55	13.58	1620.68	57.99	78.95
Q22	**5.23**	**0.25**	0.63	5.33	2.37	7.63	3.39	41.51	1.54	6.23
GM	9.97	0.70	1.43	9.44	4.20	17.08	7.59	228.84	8.33	28.37

2.2 Paper P2. Crystal: A Unified Cache Storage System for Analytical Databases

No unused data should be cached. Almost all cloud-native data warehouse systems utilize caches for cloud object storage retrieval. Given their disaggregated storage architecture, caches reduce the latency of small object requests and enhance performance by leveraging fast local NVMe SSD storage. Even on instances with high-bandwidth networking, caches help improve query throughput by combining network and local disk bandwidth. In public clouds, however, the size of the locally attached SSDs is limited. Current caching solutions operate on file- or block-level granularity, often wasting precious storage space. An optimized caching strategy that uses predicate information (i.e., filters) for storing only relevant data points may demonstrate improved query performance, which leads to the following research question:

> ***Research Question 2.*** *What performance impact does smart caching of filtered data tailored to the observed workload have on analytics, considering disaggregated storage with limited local storage?*

2.2.1 Hypotheses

Semantic caching provides better cache utilization. As mentioned, current solutions operate on the file or block level. As soon as one tuple within such a block is of interest, the entire block must be brought into the cache. Although data is often accessed in a temporal manner, a static partitioning may lead to imperfect caching. To demonstrate this issue, consider this example: A bank partitions customers according to their locations, but some analyses are performed across all customers with similar characteristics, e.g., earnings. In this example, the static partitioning does not introduce any locality, as the partition key is not aligned with the predicate. The following testable hypothesis promises better cache utilization:

> ***Hypothesis 2.1:*** *By storing filtered data based on observable query history, our cache becomes aware of the workload. This results in superior query performance compared to existing block- and file-level caches because of improved cache utilization.*

Semantic caching is adaptive while providing high query performance. The objective of caches is to improve performance and adapt to changing workloads. When query predicates change because users want to explore different aspects of their data, the cache needs to adapt. Reusing our bank example, after exploring one customer type, the data scientist analyzes another type with different earning characteristics. To cope with predicate changes, our cache needs to adapt to the workload, as summarized in the following hypothesis:

> ***Hypothesis 2.2:*** *Swiftly determining which filtered data to cache and balancing short-term and long-term knowledge improves query performance and guarantees a fast response to changing regions of interest.*

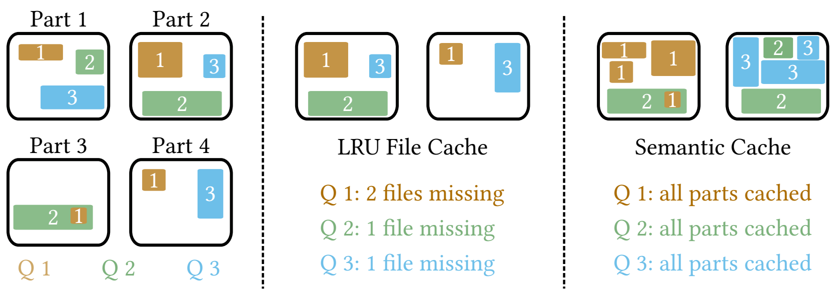


Figure 2.5: Benefit of semantic vs. traditional caching when different queries are frequently scheduled, given a maximum storage space of two blocks.

2.2.2 Scientific Method and Design

Semantic caching improves utilization. Traditional block-level caches can suffer from poor storage space utilization, particularly when only a few records per block are relevant. Figure 2.5 illustrates this for an example, which consists of three queries accessing four blocks of data. With a traditional block-level cache and a replacement strategy like LRU, none of the queries can be fully computed from the local cache. However, semantic caching, which aggregates only the necessary information for processing, enables the local execution of all queries.

Crystal considers semantic overlap and remains responsive. Our developed cache management system, *Crystal*, uses push-down predicates to cache subsets of data, called regions. These regions are views of database tables, and the content of the cache is computed with the help of semantic caching. In data warehouses, a large number of queries access similar parts of tables, often resulting in queries with overlapping predicates. *Crystal* takes this overlap into account when determining which cached data to evict, creating a complex optimization problem. *Crystal* automatically optimizes for the observed workload without any prior knowledge on the queries or data. The cache is divided into the requested region (RR) and the oracle region (OR). For the OR cache, *Crystal* determines which potentially overlapping regions should be kept in the cache. Previously uncached regions are then created and stored as Parquet file. Due to the high computational complexity, this is only executed periodically. In contrast, the RR cache is designed for new queries and can quickly adapt to workload changes. This two-region approach provides an efficient and responsive data caching solution.

Crystal's overlap-aware knapsack algorithm. The objective of the oracle is to identify the best set of regions for the observed workload, given a size budget. This problem can be viewed as a knapsack problem. Typically, knapsack prob-

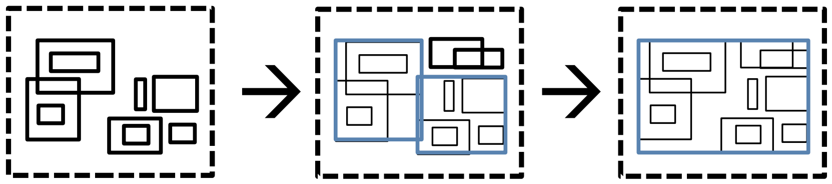


Figure 2.6: Approximative merging generalizes to the region of interest. Merged regions are marked in blue.

lems are solved either exactly through dynamic programming or approximately through greedy algorithms. Practically, the straightforward 0.5-approximation method yields effective and quick results ($\mathcal{O}(n \cdot log(n))$). This algorithm sorts items (i.e., regions) by their cost-benefit ratio and repetitively selects the next item with the highest cost-benefit ratio until the budget is reached. However, *Crystal*'s knapsack differs from the standard definition due to the dependencies introduced by overlap and the potentially reduced benefits after picking an item. Both knapsack versions are inadequate in handling such overlapping dependencies without adaptations. *Crystal*'s algorithm for the knapsack problem reassesses the cost-benefit ratio at each step of selecting a new item. Hence, the algorithm accounts for new overlap between the already-picked items and the remaining choices. This reevaluation alters the computational complexity of the greedy approximation used during *Crystal*'s cache optimization to $\mathcal{O}(n^2 \cdot \log(n))$.

Approximative merging generalizes to the region of interest. *Crystal* uses our novel approximative merging algorithm to predict future data regions of interest and generalize the cached regions. To avoid overfitting, this algorithm augments the knapsack's candidate set with previously unseen and enlarged regions. After adding these augmented regions, their cost-benefit ratio is evaluated based on historical data. During the augmentation step, overlapping or adjacent regions are merged. The resulting region's dimensions are determined by the global minimum and maximum values of the source regions. The augmentation can help to generalize to the region of interest over time, as depicted in Figure 2.6.

2.2.3 Implementation Overview

Crystal is a stand-alone cache management system. *Crystal* is a high-performance cache management system that operates as a stand-alone application, sitting between the DBMS and cloud object storage. It is written in C++ and supports various data types and query predicates. The storage format is Apache Parquet [Apa23d] and utilizes the Apache Arrow [Apa23a] library to perform read and write operations. *Crystal* uses *Gandiva* [Apa18], a newly developed execution engine for Arrow, to improve the performance of filtering tables. *Gandiva* filters are compiled into LLVM code and executed on in-memory Arrow data, resulting in a highly performant and flexible solution. *Crystal* efficiently parallelizes operations, making it suitable for low-latency DBMS middleware.

Lightweight data source connectors communicate with Crystal. *Crystal* communicates with the DBMS via socket connections and transfers data files through shared disk space or ramdisk. Since DBMSs can already process Parquet files, adapting the Parquet reader for interacting with *Crystal* is straightforward. As *Crystal* returns Parquet files, the DBMS can work with them without any code modifications. The interaction involves a minimal set of control messages, including scan request and completion messages. We test *Crystal* with two database systems – Apache *Spark* [Arm+15] and *Greenplum* [Lyu+21; VMw23]. Using only 350 lines of Scala code, we developed a new data source connector that extends *Spark*'s Parquet processing. This connector overrides the Parquet scan method to retrieve files recommended by *Crystal*. Similarly, we extended the Parquet connector of *Greenplum PXF*, an extension framework, with *Crystal* communications using less than 150 lines of code.

2.2.4 Experimental Evaluation and Conclusion

Testbed and workload. To test the hypotheses, we run experiments on a single Azure `DS14_v2` virtual machine that has 16 cores, 112 GiB of RAM, and 224 GiB of SSD storage. This instance type has a maximum network bandwidth of 12 Gbit/s. The experiments include Apache *Spark (3.0.1)*, pre-built for Hadoop (3.2), and utilize Apache Arrow (3.0.0) and azure-storage-cpplite (be490ed) as part of the software stack. All the following experiments are executed on the fact table of TPC-H (lineitem) at a scale factor of 50, stored on standard Azure Blob Storage. The query predicates use typically explored columns of lineitem. The experiments build on multiple query types that, for example, answer questions related to revenue generation. We test the caching capabilities by using a regional access pattern on lineitem, similar to prior work [Din+20]. Each query type targets a region comprising 10 - 15% of the total tuples. Individual queries read a subset of a random sample within that region, equivalent to approximately 1% of the total tuples, simulating the deep-dive of data scientists into different aspects of the data. These regions are accessed in a non-uniform distribution for more realistic user patterns and are publicly available at [Cry21].

Crystal's high cache utilization improves query performance. Hypothesis 2.1 claims that semantic caching can utilize storage space more efficiently. To test this hypothesis, we compare *Crystal* with *Alluxio* [All23; Li+14], a state-of-the-art block-level caching solution for Apache *Spark*. Figure 2.7 visualizes

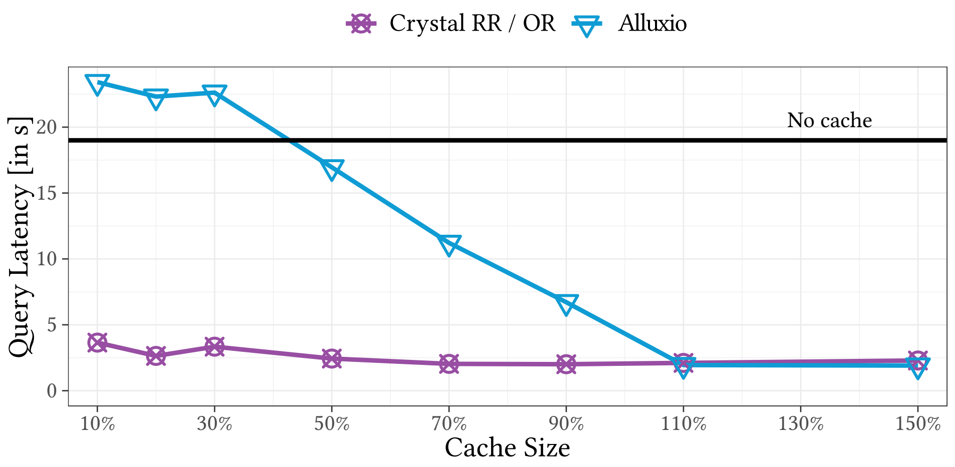
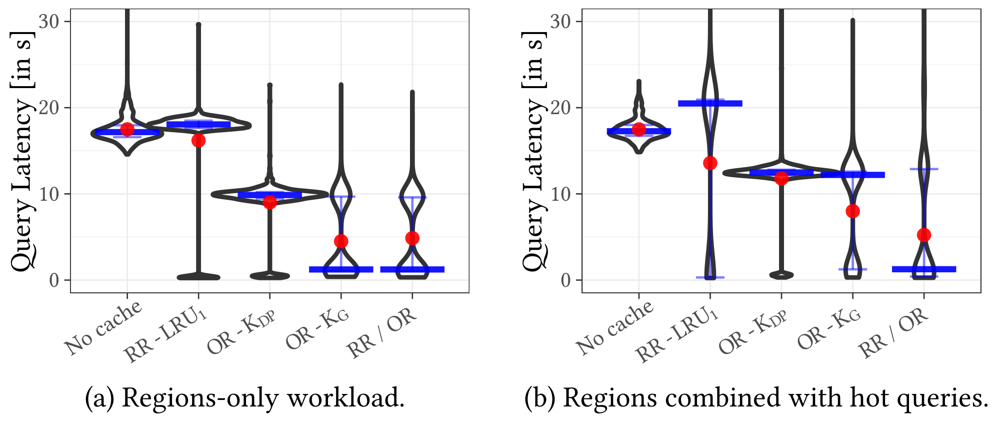


Figure 2.7: Crystal's semantic regions outperform Alluxio's caching strategy.



(a) Regions-only workload.

(b) Regions combined with hot queries.

Figure 2.8: TPC-H workload with different caching strategies.

that *Crystal* exhibits robust performance, even with limited storage space. On the other hand, *Alluxio*'s performance increases as more caching space becomes available but lags behind *Crystal*'s performance until all data is cached. In Figure 2.8a, we test various caching strategies in *Crystal* on the same workload. We restrict the caching space to 20% of the original data size. While *RR-LRU*$_1$, a block-level LRU-based caching strategy, fails to enhance performance, our fast and overlap-aware greedy algorithm *OR-K*$_G$ significantly improves query latency. Although short-term knowledge is not helpful in this experiment, our proposed *RR-LRU* / *OR-K*$_G$ combination has a performance similar to *OR-K*$_G$. These experiments confirm Hypothesis 2.1.

Crystal adapts quickly to changing workloads. To demonstrate workload adaptivity as stated in Hypothesis 2.2, we change the region of interest during the

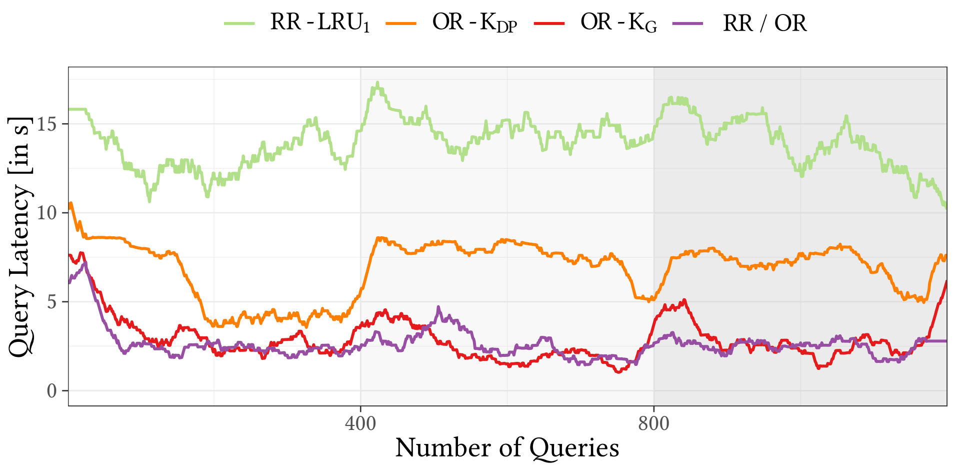


Figure 2.9: Adaptivity to changing regions of different caching strategies.

workload execution in Figure 2.9. After 400 queries, the query focus is changed to a different region. Our overlap-aware greedy knapsack algorithm quickly copes with these changes and continues to provide excellent cache utilization. In contrast to the greedy version, the dynamic programming knapsack is slow in adapting to new workloads due to its high computational complexity. A pure LRU-based block-level algorithm would adapt quickly but cannot improve query performance of the region workload due to inefficient cache utilization. However, during hot queries such as the latest news on a news website, a small LRU-based cache (RR) can improve performance, as depicted in Figure 2.8b. Both experiments show that *Crystal*, using *RR-LRU* / *OR-K_G*, adapts quickly to changes in the workload, validating Hypothesis 2.2.

High-performance and adaptive caching on limited disk space. The experimental evaluation shows that *Crystal* has better cache utilization than current caching solutions. Our overlap-aware knapsack algorithm, combined with our novel approximative merging of neighboring regions, facilitates high query performance, even with fluctuating query patterns. *Crystal*'s semantic regions have proven to be especially useful in environments with limited local storage space. Moreover, the small short-term knowledge cache ensures robustness against predicate spikes in the workload.

2.3 Paper P3. JSON Tiles: Fast Analytics on Semi-Structured Data

JSON is used commonly, but analytics is expensive. In recent years, there has been a growing trend towards storing data in a semi-structured format. Every document contains the structure of the data, reducing API complexity. Unfortunately, the structural information in each document is harmful for analytical performance. As every tuple can, in principle, contain arbitrary values, scanning a raw or binary-optimized JSON document for a single value involves an expensive lookup within each document. Although the structure can be arbitrary, most of the time, JSON is machine-generated, and, therefore, documents from the same system have similar components. For example, documents from logging services use similar keys, such as error code and message, even for different types of errors. We want to exploit this structural similarity to improve JSON document processing in relational database systems, as stated by the following research question:

> ***Research Question 3.*** *How can database systems efficiently infer and exploit the structural information of a set of documents to enable fast and robust analytics on semi-structured data?*

2.3.1 Hypotheses

Exploiting structural components improves JSON processing. Database systems usually store JSON data in a column, where each cell represents a full JSON document. These columns often use a textual or binary representation of JSON. Accessing values within a JSON document involves at least one order of magnitude more CPU cycles compared to a pure value scan of a column-oriented database. By using the internal schema of JSON data and extracting columns of the important values, significant performance improvements are expected, similar to relational scans. Structural component changes within documents often occur over time. When storing documents of the same type, document changes over time are represented in the insertion order. Extracting columns on only a small subset of documents (a tile) allows us to generalize to these changes. These properties are summarized in the following hypothesis:

> ***Hypothesis 3.1:*** *Exploiting the implicit schema of a small subset of JSON documents provides significant improvements in query performance when analyzing stored data and robustness to document changes over time.*

Reordering achieves robustness beyond the insertion order. Although document changes are often aligned with the insertion order, it is not unlikely that multiple different document types are stored within the same column, exploiting JSON's ability to store specialized information for each individual tuple. Even if multiple document types are inserted randomly, the database system needs to be able to provide fast analytics on these documents. However, the low frequency of similar structures within a single tile prohibits the extraction of columns. Reordering of documents between tiles helps to group similar documents together, increasing the frequency of common structures and enabling materialization of randomly inserted heterogeneous document types. The following hypothesis summarizes the robustness objective:

> ***Hypothesis 3.2:*** *Reordering documents guarantees robustness and handles random insertion of heterogeneous document types without sacrificing the insertion performance.*

2.3.2 Scientific Method and Design

Improvements for JSON processing. *JSON tiles*, our proposed collection of algorithms and methods, addresses several challenges in managing JSON data. First, it enhances access performance by storing JSON data in a columnar representation, enabling fast scans. This is a considerable improvement over standard JSON access that involves slow object traversal for every JSON document. Second, *JSON tiles* maintains data statistics by collecting information about individual keys during data loading. This allows for better query optimization, particularly for complex multi-table queries. Third, it offers robustness for handling heterogeneous data with varying document types, changes in fields over time, and previously unseen data. It accomplishes this through local computations, reordering of documents, and dealing with outliers effectively.

JSON tiles extraction algorithm. *JSON tiles* differs from the global schema extraction presented in *Sinew* [TDA14] by detecting implicit document structures at a fine granularity. Instead of attempting to extract a single global schema, *JSON tiles* divides the input data into smaller tiles, each containing a local schema. This approach takes advantage of the spatial locality found in many datasets. For example, changes in the document structure over time is usually aligned with the insertion order. Thus, extracting the new structural components in subsequent tiles. The extraction process in *JSON tiles* involves three main steps, demonstrated with a simplified example consisting of eight documents from Twitter and a tile size of four, depicted in Figure 2.10:

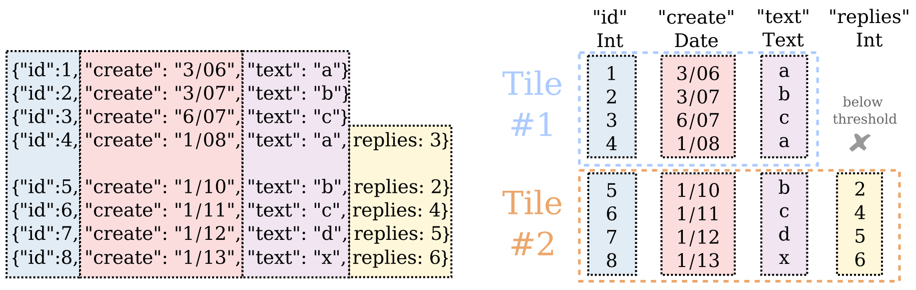

{"id":1, "create": "3/06", "text": "a"}
{"id":2, "create": "3/07", "text": "b"}
{"id":3, "create": "6/07", "text": "c"}
{"id":4, "create": "1/08", "text": "a", replies: 3}
{"id":5, "create": "1/10", "text": "b", replies: 2}
{"id":6, "create": "1/11", "text": "c", replies: 4}
{"id":7, "create": "1/12", "text": "d", replies: 5}
{"id":8, "create": "1/13", "text": "x", replies: 6}
"id" Int
"create" Date
"text" Text
"replies" Int
Tile #1
Tile #2
below threshold

Figure 2.10: Materialization example on simplified Twitter data, using a tile size of four and a frequency threshold of 75%.

1. For each tuple, all key paths are collected. A key path represents the full path from the document root to the actual key-value pair, similar to an absolute path in file systems. In this example, the first three tuples have three key paths { id , create , text }.
2. Frequent itemset mining is applied to the collected key paths [AS94; HPY00]. We determine the frequency of each itemset and prune itemsets below the extraction threshold. From the remaining itemsets, we identify the maximum itemsets. The only maximum itemset of the second tile is ({ id , create , text , replies }, 4).
3. *JSON tiles* extracts the union of these maximum itemsets, such that the included key paths are materialized as relational columns.

Reordering improves robustness. After inserting enough tuples to reach the tile size, a new tile is created. Therefore, the content of the tiles is based on the order in which tuples are inserted. Although many datasets exhibit spatial locality consistent with the insertion order, random insertion of multiple document types is also common. Random insertions likely prohibit the materialization of columns as the extraction threshold cannot be reached. To address this issue, *JSON tiles* applies reordering to cluster tuples with common frequent itemsets into the same tile, regardless of their original order. This reordering process guarantees that more tiles fulfill the extraction threshold and can be materialized efficiently. Tuples are matched according to the previously detected itemset and grouped accordingly. After the clustered tuples are combined in one tile, the original extraction algorithm identifies the columns to be materialized. In conclusion, this reordering technique improves the organization of data within *JSON tiles*, resulting in more data materialization for workloads with little spatial proximity. This reordering is visualized in Figure 2.11, in which each cell represents a tuple, and the texture highlights the best-matched itemset of this tuple.

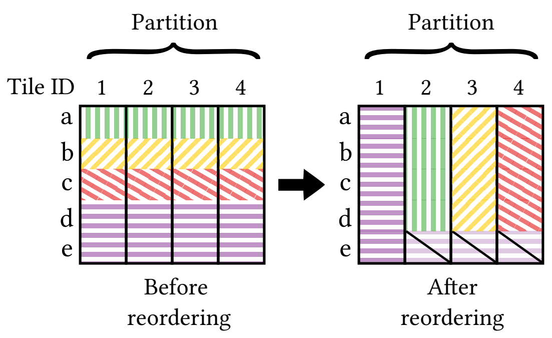


Figure 2.11: Better materializable tiles by reordering of documents.

Query optimization with aggregated statistics. *JSON tiles* improves query optimization for JSON data by providing per-key statistics and estimators. During the creation of tiles, additional information is collected for each individual tile. This data is aggregated at the relation level and then used for query optimization. This is particularly important for better cardinality estimations during join ordering [Lei+15]. *JSON tiles* stores frequency counters and HyperLogLog sketches for each frequent key path [Fla+07]. These distribution estimates enable reasoning on join and table predicates. Additional sampling of documents improves estimates by evaluating query predicates on these samples.

2.3.3 Implementation Overview

Integration and storage of semi-structured documents. *JSON tiles* is implemented as part of *Umbra*, written in C++. The main part of *JSON tiles* is its storage engine that handles the extraction and reordering of JSON documents. Every tile requires a header that describes the materialized data. This header stores information on the key paths, the corresponding value types, and statistics for fine-grained optimizer decisions. Since the headers of individual tiles differ in size, only a pointer offset is kept in the fixed-size relational storage. The header data and the extracted columns are stored as variable-sized data, similar to storing long strings.

Extracted columns need predicate push-down and value casts. Because syntactically JSON documents are stored within a column of the opaque type JSON, accessing values within the JSON documents requires an extended syntax. *Umbra* employs the *PostgreSQL* syntax to access values within documents [Pos23a]. These accesses are often performed by a join or group-by operator to extract useful information from the document for joining or aggregation purposes. However, we can only benefit from the materialized columns at the scan operator. If the information is not passed to the scan operator, this

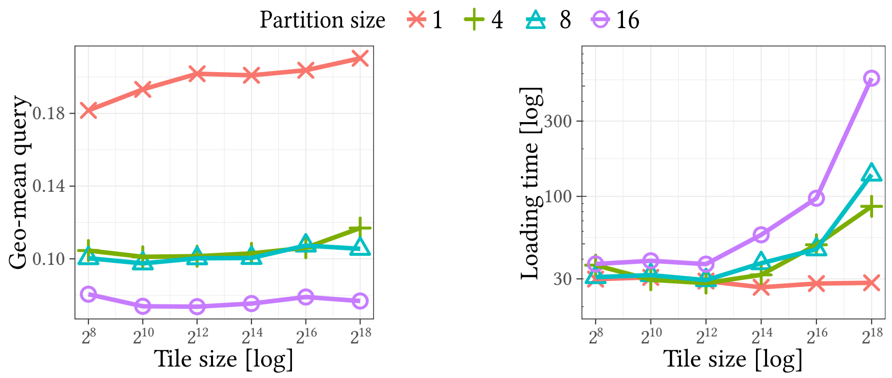
Table 2.3: Execution times of TPC-H queries for different industry-grade systems and internal strategies (in seconds).

	PG.	Spark		Hyper	Umbra			
		Mongo	Parquet		JSON	JSONB	Sinew	Tiles
1	5.276	14.297	1.939	1.950	1.725	0.178	0.122	**0.030**
2	> 100	23.383	2.735	1.370	1.608	0.584	0.637	**0.035**
3	17.905	15.892	1.288	0.560	0.675	0.280	0.259	**0.030**
4	3.013	10.439	1.755	0.539	0.692	0.227	0.228	**0.026**
5	87.468	22.659	2.072	> 100	1.340	0.372	0.326	**0.045**
6	1.259	14.896	0.690	0.244	0.254	0.119	0.085	**0.010**
7	> 100	21.035	2.554	3.111	1.177	0.429	0.351	**0.103**
8	> 100	26.608	1.814	1.156	1.469	0.474	0.416	**0.062**
9	> 100	23.688	3.939	1.728	2.576	0.395	0.370	**0.153**
10	> 100	21.967	2.003	0.984	1.362	0.388	0.294	**0.067**
11	> 100	23.444	0.809	0.829	1.070	0.344	0.353	**0.068**
12	1.493	18.783	1.316	0.419	0.450	0.286	0.289	**0.061**
13	5.570	10.597	2.146	0.683	0.665	0.149	0.291	**0.044**
14	1.502	9.552	0.734	0.343	0.392	0.171	0.142	**0.017**
15	9.105	19.024	1.306	0.339	0.399	0.211	0.185	**0.018**
16	4.220	15.119	2.693	0.898	0.629	0.201	0.273	**0.048**
17	> 100	16.379	1.381	0.605	0.567	0.173	0.091	**0.026**
18	86.167	14.861	1.849	1.388	0.949	0.260	0.179	**0.050**
19	1.290	33.885	0.970	0.363	1.834	0.213	0.170	**0.057**
20	> 100	20.234	1.613	0.787	0.974	0.355	0.348	**0.042**
21	12.372	39.236	3.517	1.415	1.787	0.615	0.479	**0.103**
22	2.060	11.306	3.135	0.529	0.566	0.172	0.180	**0.016**

operator has to produce the full JSON value and push it to the parent operator. To resolve this issue, we employ a strategy of pushing the JSON access down into the scan operator during query optimization. Parent operators replace the access with placeholders that are later filled with the values from the materialized columns. Similar to the access information, the final value type can be used to avoid intermediate value transformations. While *Umbra* stores the materialized columns with different value types, the access syntax strictly returns a textual or JSON value. To minimize the need for expensive transformations (e.g., casting text to date), we push type casts into the scan operator such that the scan directly produces the correct type.

2.3.4 Experimental Evaluation and Conclusion

Testbed. *JSON tiles* is integrated into *Umbra*, our relational database system. We compare *Umbra* (with different JSON strategies) with other industrial-strength



(a) Geometric mean performance.

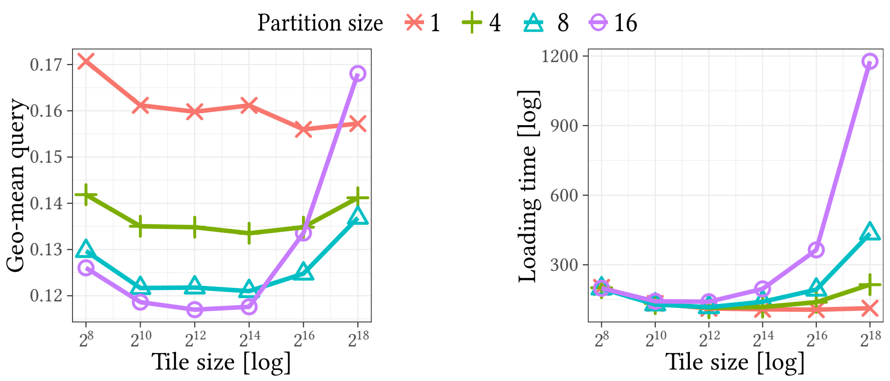
(b) Loading time.

Figure 2.12: Queries on a shuffled TPC-H dataset.

database systems, such as *PostgreSQL (12.4)*, Tableau *Hyper (0.11556)*, and Apache *Spark (3.0)* with Apache Parquet *(Dremel)*. Additionally, we compare Apache *Spark* with *MongoDB (3.6)*, where we employ *Spark* for query handling. The experiments are conducted on an AMD Ryzen Threadripper 1950X system, which has 16 cores, 32 threads, 64 GiB of RAM, runs Ubuntu 20.04, and utilizes a 2 TiB Samsung 850 Pro SSD.

JSON tiles improves analytical query performance. Hypothesis 3.1 assumes that exploiting the structural information and similarities among JSON documents enhances query performance. We evaluate this using a modified TPC-H workload, where each record of the TPC-H tables is represented as a JSON object utilizing the column names as keys. The documents from all tables are merged into a unified table using a single JSON column. Table 2.3 shows the query latencies of the adapted 22 TPC-H queries. *Umbra* with *JSON tiles* is at least one order of magnitude faster than existing industry-grade database systems. Internal comparisons show that *JSON tiles* outperforms all other strategies by at least 5× on average, validating Hypothesis 3.1.

Reordering enables robustness beyond the insertion order. Unlike past approaches that materialize JSON data, using *JSON tiles* allows for tile reordering, resulting in improved robustness and a more homogeneous extraction, as stated in Hypothesis 3.2. To demonstrate the advantages of tile reordering, we evaluate *JSON tiles* with different partition sizes and tile sizes using a completely randomized TPC-H workload of shuffled documents in Figure 2.12 and a real-world Twitter dataset in Figure 2.13. The size of the partition defines the number of tiles involved in local reordering. We further analyze the insertion time of the two datasets to argue about the cost of reordering. The experiments show that a medium partition size of eight and small tile sizes of $2^{10} - 2^{12}$ improve query performance significantly while preserving the insertion time. The partition



(a) Geometric mean performance. (b) Loading time.

Figure 2.13: Queries on a real-world Twitter dataset.

size is directly correlated to the minimum number of documents needed per each type to meet the extraction threshold. Thus, a medium size of eight decreases this minimum frequency substantially, allowing for the materialization of most initial tables from the shuffled TPC-H. The use of our block skipping optimization in combination with *JSON tiles* results in low query latencies even for heterogeneous workloads, proving Hypothesis 3.2.

Fast and robust analytical JSON processing. In conclusion, *JSON tiles* is the first approach to facilitate fast and robust analytical query processing of JSON documents while enabling advanced query optimization and join ordering. Our experiments demonstrate that *Umbra* with *JSON tiles* reduces query latency by one order of magnitude compared to industry-grade database systems. Because statistics are collected for individual keys, large join queries can be efficiently optimized and processed. Reordering guarantees robustness, even in workloads with random insertion order.

2.4 Paper P4. Experimental Study of Memory Allocation for High-Performance Query Processing

Memory allocation is on the critical path. Analytical queries can produce intermediate results that are even larger than the input. To efficiently handle large datasets, instances are equipped with faster and more CPU cores. Since accommodating more cores on a single chip is not always possible (e.g., due to heat distribution), many servers have multiple chips installed in different sockets. Main memory is physically connected to one socket, but all sockets can access all data via interconnects (e.g., QPI). All of this makes memory allocation more complex and raises the question of how strongly the chosen memory allocator affects query performance, explored in the following research question:

> ***Research Question 4.*** *What impact does the selection of the memory allocator have on query performance, scalability, and memory efficiency of analytical database systems?*

2.4.1 Hypothesis

The choice of memory allocator impacts performance. During memory allocation, the allocator requests either new memory from the operating system or uses currently unused memory it requested previously. The allocator might also decide to request more memory than needed to keep memory quickly available, sacrificing memory utilization. Deallocations either return memory to the operating system or keep it allocated to the process and reuse it for future allocations. Because memory allocators use different strategies to perform the mentioned operations, the characteristics of dynamic memory allocators (performance, scalability, memory fairness, and memory efficiency) differ, which is stated with the following hypothesis:

> ***Hypothesis 4.1:*** *Choosing the right memory allocator is crucial for analytical query performance, scalability of the database system, memory fairness to other processes, and memory efficiency.*

2.4.2 Scientific Method and Design

Design of the memory allocation study. Memory allocation is closely tied to the operating system, as it manages the mapping between physical and virtual memory. Allocators must request virtual memory from the OS. The allocator either uses already held memory or requests new memory from the OS, which can be achieved through APIs provided by the kernel. The mechanism of requesting and holding memory differs between various allocators. To determine the best allocator in terms of performance, scalability, memory fairness, and memory efficiency, we perform extensive experiments with five different memory allocators within our modern database system *Umbra*.

Allocation pattern for decision support workloads. Our analysis of the allocation pattern for TPC-DS [NP06] in *Umbra* reveals that the majority of allocations are between 32 KiB and 512 KiB. In particular, tuple materialization of group-by and join operators contributes most medium-sized allocations. Additionally, larger memory regions are allocated for holding the bucket arrays of

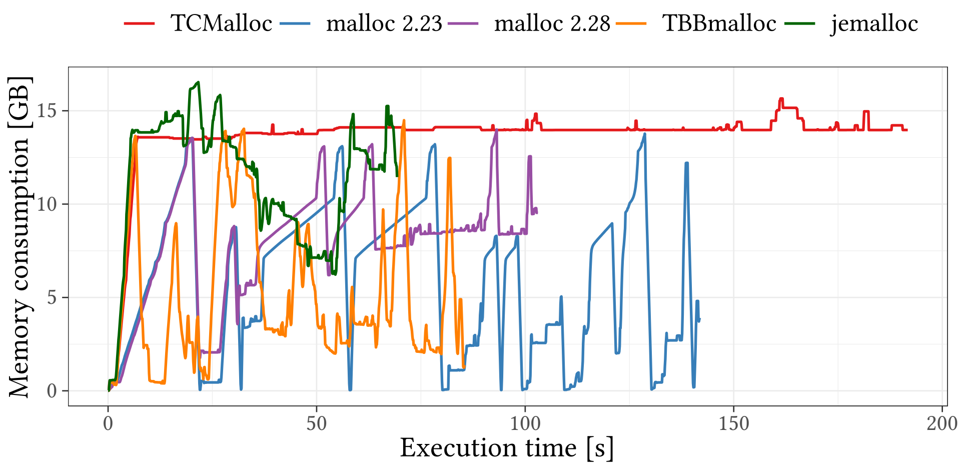


Figure 2.14: Memory utilization during single query execution of all TPC-DS queries on the 4-socket Intel Xeon server.

chaining hash tables. Note that *Umbra* employs a transaction-local chunk allocator for small allocations (< 32 KiB) to improve transactional query throughput.

More realistic query patterns with exponential distributions. To create a realistic workload, we use an exponentially distributed query arrival model. The introduced variability in the number of concurrently active transactions simulates workload variations and results in a more complex allocation pattern. The different query types are chosen uniformly, but the starting times follow an exponential distribution. Events arrive at an expected interval of $1/\lambda$ and with a variance of $1/\lambda^2$.

2.4.3 Implementation Overview

Details on collecting memory statistics. A collection of methods is integrated into *Umbra* to monitor memory statistics and allocations. Memory statistics are polled from the kernel at short intervals during query execution. To retrieve the statistics, we read the kernel's virtual memory files located in `/proc`. Additionally, we track the allocated memory within the database's memory pool per operator. These statistics can be collected with minor performance implications. To track memory usage patterns, additional tracing of every memory allocation call can be enabled. This additional allocation call tracing is expensive and disabled by default to avoid impacting benchmark results. A new driver executes the exponentially distributed workload and allows us to precalculate the query scheduling times for deterministic executions.

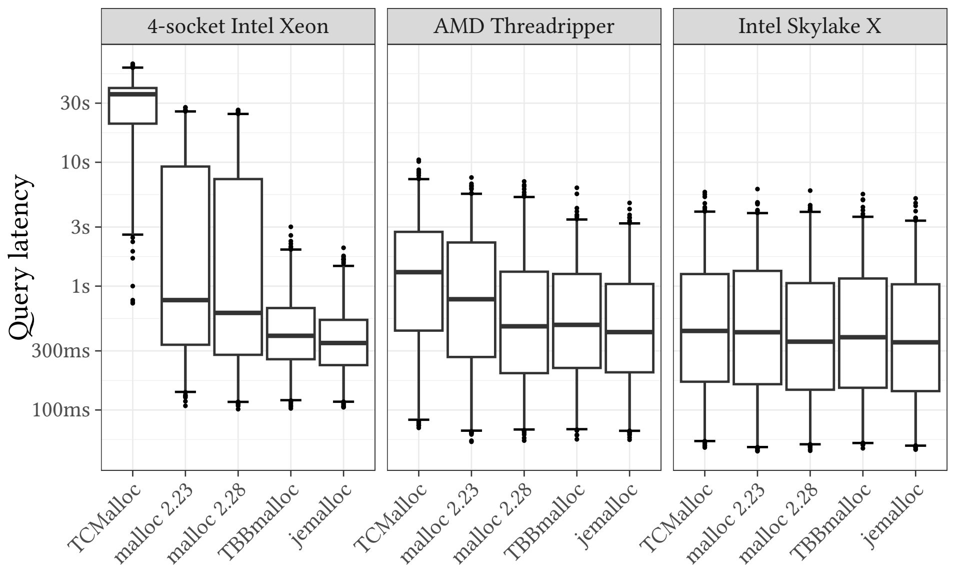


Figure 2.15: Memory allocator scalability on different systems using the exponentially distributed TPC-DS workload (λ = 6 q/s, SF 10).

2.4.4 Experimental Evaluation and Conclusion

Memory allocation has performance and memory usage implications. We conduct experiments that evaluate the impact of memory allocation to verify Hypothesis 4.1. The experiments use the TPC-DS analytics benchmark with a scale factor of 100. Figure 2.14 shows a sequential execution of all TPC-DS queries in *Umbra* utilizing five allocators: glibc *malloc 2.23*, glibc *malloc 2.28*, *jemalloc 5.1*, *TBBmalloc 2017*, and *TCMalloc 2.5*. Surprisingly, this straightforward workload exhibits significant performance and memory usage differences. For instance, the runtime is reduced by half when comparing *jemalloc* with glibc *malloc 2.23*. On the other hand, *malloc 2.23* tends to return memory earlier, making it a fair allocator to other processes. The memory currently allocated is an upper limit because any memory freed with `madvise`, a new kernel method to return memory, is not accounted for due to its high runtime overhead. While *jemalloc* and *TCMalloc* use this feature, only *TCMalloc* returns all memory to the operating system. Scalability is assessed using a more practical workload with arrival times that follow an exponential distribution. Figure 2.15 illustrates that with larger instances higher differences are observed. In particular, the 4-socket Intel Xeon server shows significant performance differences. These experiments confirm Hypothesis 4.1.

Huge differences between allocators. Our experimental study finds significant differences among five tested memory allocators in terms of memory fairness, efficiency, scalability, and query performance. Extensive analysis of TPC-DS workloads indicates that *jemalloc* is the top-performing allocator. As a result, *jemalloc* is adopted as the default allocator for *Umbra*.

CHAPTER 3

Related Work

Rich line of analytics research. Due to the increasing demand for analytics solutions in the cloud era, a plethora of new approaches and systems have been developed to handle different storage architectures and heterogeneous data [Adr22]. Ample research in cloud processing facilitates building cloud-native database systems.

High-Performance Analytics on Cloud Object Storage

Database systems adapt a cloud-native design. Software-as-a-service (SaaS) database management systems often specialize in catering to either OLTP [Ant+19; Cao+22; Cao+21; Ver+17] or OLAP [AR20; Beh+22; Dag+16; Mel+10; Mel+20; Pan21; Set+19; Van+18; Vup+20; Zha+19] requirements, addressing the unique challenges posed by the cloud environment. For example, AWS provides *Aurora* [Ver+17] for transactional workloads and *Redshift* [Arm+22] for analytical workloads. A survey of Alibaba shows that their current products are usually assigned to one of these two tasks [Li19]. Although most systems store their data on cloud storage, their strategy is to cache most of the data on local nodes. An experimental study thoroughly examines and contrasts the architectural distinctions between these systems [Tan+19a]. Our system, *Umbra*, would be classified as an engine that can process data directly from remote storage. Since cloud object storage is inexpensive, it is used as a basis for durable storage solutions. Two examples are Apache *Iceberg* and *Data Lake* [Apa23b; Arm+20]. By using metadata, which is stored alongside the data, these data stores are able to provide snapshot isolation. Since *Umbra* also stores all metadata in cloud object storage, these storage backends can be easily integrated.

Challenges and opportunities in cloud data processing. Early work discusses challenges of processing data directly from Amazon S3 for OLTP use cases [Bra+08]. An experimental study compares the different hardware resources of the AWS EC2-provided instances [SDQ10]. Since instances have different dollar-per-resource characteristics, recent work investigates instance selection to optimize cost and performance for a given workload [LK21]. *Pix-*

els [BA22] describes the high retrieval latency of data from cloud object storage as a major challenge. It applies optimizations on the columnar PAX layout such that columns are ordered according to their access frequency, and neighboring PAX blocks are merged. Combined with IO scheduling optimizations that exploit these storage properties, the number of requests can be reduced, resulting in lower data latencies. Using these storage optimizations is orthogonal to our developed low-overhead client *AnyBlob* and our task-based retrieval integration in *Umbra*.

Cost savings with spot instances. Due to substantial discounts, research has been conducted to address the issue of spot instance termination [SI17; Sub+15]. One such solution is spot instance hopping, which relies on the cloud vendors' notification before the instance is taken away. Because *Umbra* has no issues with cold starts on network-optimized instances, it is particularly well-suited for this strategy. While *Umbra* processes queries (TPC-H, scale factor 1000) faster than the AWS termination delay [Ama23h], research shows the feasibility of live migration of query states to overcome full query restarts at new instances [Win+22].

Analytics on serverless functions. To further increase flexibility, cloud vendors provide serverless compute functions, which are executed on-demand without specifying an instance. Although serverless functions provide cost and maintainability benefits for short-running applications, steady-state workloads are more expensive. Additionally, serverless functions come with many hardware restrictions. They can only run for a limited amount of time and only come with little memory, a small number of CPU cores, and low-bandwidth networking [Hel+19]. Recent work proposes solutions to overcome the aforementioned hardware restrictions and demonstrate analytics on serverless functions [BSA23; MMA20; Per+20]. Although these systems research S3 retrieval in serverless environments, the retrieval characteristics differ significantly between low-bandwidth cloud functions and high-bandwidth EC2 instances.

Data center developments. Data centers are anticipated to be equipped with high-throughput (1 Tbit/s) Ethernet connections in the following years [Cai+22]. A study focusing on kernel storage APIs has identified io_uring, utilized in *AnyBlob*, as a promising technology [Did+22]. Particularly for NVMe SSDs, io_uring is gaining widespread adoption and use [Did+22; HHL20; Lei+23; LB21; Par+21]. Future data centers may additionally decouple memory from computing, independent of the current storage disaggregation [Kor+22; Wan+22; Zha+20; Zha+21].

Caching in Disaggregated Storage Environments

Resource-conscious caching is useful for many workloads. Due to high access latencies from external storage, caching is still vital for workloads not dominated by data bandwidth. Although high-bandwidth networks are becoming more affordable, not all cloud vendors provide instances with 100 Gbit/s networking. Even on network-optimized instances, caching improves analytical query performance by utilizing the additional bandwidth to local storage. Because the caching space on the local node is limited, resource-conscious caching is helpful for many database systems.

Building on ideas from semantic caching. Semantic caching, initially introduced in *Postgres* [Sto+90], has seen subsequent enhancements through extensive research [CR94; Dar+96; Des+98; KFD00; KR99; SSV96; SSV99]. There, query results are cached to expedite recurring queries. The basic idea of the caching approach within *Crystal* builds upon this rich line of work. Semantic caching autonomously selects cached views while considering various factors like size, access frequency, and materialization cost, often relying on cost-based policies. Unlike previous work on semantic caching that caches entire query results, *Crystal* caches only intermediate results of query selection and projection operators. While caching complete query views benefits repeated queries, it reduces view reusability. Another differentiator is that most research does not explore overlapping views. Although some research [Dar+96; Des+98] explores overlapping, they propose to split queries into non-overlapping segments. However, this may lead to a large number of small views and increased processing overhead. Chunk-based semantic caching [Des+98] addresses this by dividing the hyper space into independent regions. Because this division is static, this approach lacks adaptability to unknown query patterns. Modern cloud storage solutions allow querying data given a predicate. New database architectures [YLH23; Yan+21; Yu+20] rely on this feature; however, this push-down of predicates to remote storage is expensive. Nonetheless, our predicate-aware cache is well-suited for pushing down predicates to cloud storage.

Managing and storing intermediate results for accelerated queries. Materialized views store query results in separate tables [GL01; SDN98; Sri+96; Zho+07]. In contrast to *Crystal*, users typically define views for caching or materialization manually, often done by a DBA. Besides engineering efforts, materialized views demand advanced query optimizer algorithms to determine if a query can use a materialized view and when to update it after changes to the base table. Several techniques focus on reusing intermediate query results instead of final results. Some methods [TGO01; Tan+19b] share intermediate results among concurrent queries. Other approaches [Dur+17; Iva+09; Jin+18a;

Jin+18b; NBV13; PJ14] store intermediate results for reuse by subsequent queries and employ a replacement policy for evicting intermediate results when size limits are reached. However, integrating these intermediate result caching techniques into a DBMS is complex, whereas *Crystal*'s base table caching can be utilized with lightweight and easy-to-implement database-specific connectors.

Mid-tier caching reduces workload at the ground-truth database. In multi-tier database setups, mid-tier caches reduce backend database workloads [Alt+03; Bor+04; LGZ04]. These caches maintain shadow databases to store query results, but users must usually define cached views manually. Vendors offer various solutions to cache data on SSDs, such as *Databricks Delta Cache* [Arm+20], *Alluxio* [All23], and *Snowflake Cache Layer* [Dag+16]. These solutions cache files at the page or block level and leverage common replacement policies like LRU. In comparison, *Crystal* serves as a versatile caching layer by caching data in a more efficient layout (reorganizing rows based on queries) and format (e.g., Parquet), accelerating subsequent query processing. Modern formats allow pruning data by push-down predicates, optimized with bit manipulations in recent work [LLC23]. Different encoding schemes and compression algorithms are used to improve the storage format [Kus+23]. Moreover, the challenges in cloud storage fostered work on ephemeral storage systems [Kli+18] and multi-tier storage solutions [Yan+21; Zha+22; ZBL22].

Fast Analytics on Semi-Structured Data

JSON processing in database systems. The rising popularity of JSON among developers forces relational database systems to invest in handling semi-structured data formats. Database systems either store JSON data in their relational tables as an additional column or use the raw JSON files to process queries. Both solutions also allow indexes for faster processing of JSON documents [CLP13; KAA16; LG15; LHM14; Tab23].

Sinew uses coarsely grained chunks. *Sinew* [TDA14] is a *PostgreSQL* extension that is able to detect frequent document fields on an entire table granularity. These fields are extracted to speed up analytical processing. However, *Sinew* has robustness issues when dealing with changing or combined data due to the coarse detection granularity. In contrast to *Sinew*, our system focuses on providing robust extracted relational chunks, even when heterogeneous document types are randomly inserted.

Proteus produces customized code and leverages indexes. *Proteus* [KAA16] generates code for accessing heterogeneous data. It creates structural indexes when it first loads JSON documents to improve subsequent access performance on the raw JSON documents. *ReCache*, which is an extension of *Proteus*, accelerates the processing of JSON documents by introducing cost-based caching of heterogeneous data. This strategy monitors the query workload and uses rules to determine the best memory layout of the cached data [AKA17].

Open-source formats used for JSON. Other systems, such as Apache *Spark* [Arm+15; Zah+16] and *Hive* [Thu+09; Thu+10], employ storage plugins for handling heterogeneous data. They rely on common open-source formats for storing JSON data, like Apache Parquet [Apa23d] and Avro [Apa23c]. Apache Parquet relies on record shredding, described in *Dremel* [Mel+10], to robustly store JSON data. However, *Spark*'s query performance is slower than approaches that extract data directly due to the high overhead of the access automata. To improve query throughput in database systems using Parquet files, previous work demonstrates how to better parallelize them [RFN23]. However, connecting the nested and repeated levels is not explored in this work, which is needed for general-purpose JSON documents. Another solution is to represent JSON data with a lossless tree encoding, which helps find tuples according to their edit distance. Analyzing the edit distance is useful for similarity-based duplicate removal [Hüt+22].

Raw JSON processing avoids insertion. Instead of storing columnar JSON data, raw files can be queried directly [Ala+12; Bla+14; CR14; Pav+18]. This approach avoids the explicit insertion into the database system. The idea is that queries can be specified over raw data files, which are then executed without any loading delays [Idr+11]. The raw data access necessitates a fast data parser. Parsers such as *FAD.js*, *Mison*, or *SIMD-JSON* use modern CPU features, e.g., SIMD, for fast reads that allow to saturate disk bandwidth [BB17; Ge+19; LL19; Li+17]. These approaches can be combined with raw filters to speed up parsing or reduce the amount of ingested data [Pal+18; Xie+19]. Optimizations for handling distributed processing on JSON files help to improve performance [SEZ23]. Even specialized hardware (FPGA) is considered to optimize JSON parsing [Dan+22].

Document stores use JSON internally. Document stores, such as *MongoDB* [Mon23b], *Couchbase* [Cou23], and *DocumentDB* [Ama23i], build on the usage of semi-structured data internally. As these document stores are built for point accesses, analytical queries are slower compared to relational systems.

JSON description language infers schema. Due to the progress in building a description language for JSON, known as *JSON Schema* [JSO20], recent work studies the inference of the schema within documents [Baa+17; DA16]. These schema file computations, however, are very CPU intensive because a detailed list of optional and required schema fields needs to be inferred.

Impact of Memory Allocation on Query Performance

Memory allocation must be reevaluated given the current architecture. Ferreira et al. conducted an analysis of dynamic memory allocation in multi-threaded workloads [Fer+11]. Although the study analyzes important factors, it only considers multi-threading up to 4 cores. In contrast to today's deployments, where systems like *HANA* [MBL17] utilize hundreds of general-purpose cores on a single instance, the number of parallelism is too low for a meaningful recommendation. Modern analytics systems are built on top of allocation-critical operators such as hash joins and aggregations. Therefore, a revised evaluation is needed in this changed environment. Note that systems often implement the aforementioned operators in slightly different flavors [Bal+13; BLP11; Lei+14; ZDP19]. Thus, the allocation pattern is affected by these design choices.

Chunk allocation benefits transactional queries. OLTP systems optimize performance by managing memory allocations in chunks, which benefits short-running transactional queries [DN19; SA13; Tu+13]. However, many database systems also handle analytical workloads. Consequently, different memory allocation patterns must be considered. Custom memory managers that allocate larger chunks of memory can improve performance but at the expense of memory efficiency. Using a coarser granularity reduces the number of memory calls needed for smaller allocations. Therefore, *Umbra* uses transaction-local chunks to improve the performance of small allocations. Despite this optimization, allocations remain a performance problem. This emphasizes the importance of choosing the right allocator to maximize throughput.

CHAPTER 4

Conclusions

Improvements for cloud-based data processing. The shift from on-premises to cloud-native database systems introduces several distinct challenges and opportunities. This dissertation's publications introduced various methods to help efficiently and cost-effectively process large data quantities in the era of cloud computing.

High-performance analytics on cloud object stores. The emergence of fast interconnects are facilitating the separation of storage and compute. To best utilize disaggregated storage, database systems must transition from a disk-centered design to managing network storage. We presented an experimental study that analyzed the characteristics of cloud object storage for high-performance analytics. With the results, we developed *AnyBlob*, a novel low-overhead multi-cloud retrieval library. Our findings showed that *AnyBlob* can reduce CPU usage by 30%, leading to more available resources for analytics. Through careful integration into database systems, we demonstrated the cost-effectiveness and high performance of analytics on cloud object storage. Our database system reduces the cost compared to cloud offerings by almost an order of magnitude while maintaining similar performance, even when caching is disabled.

Adaptive semantic caching improves query performance significantly. To address the increased latency and bandwidth limitations of network storage, a typically employed solution is local disk caching. Even with today's fast networks, caching increases the total instance throughput by combining network and disk throughput. Traditional caches use an eviction strategy comparable to LRU and store cached data on a block or file level. However, this can lead to poor storage space utilization if the data is not partitioned in accordance with the accessed query predicates. Semantic caching can overcome this issue by storing only necessary information, which is particularly important on public cloud instances where SSD space is limited and expensive. We demonstrated the feasibility of semantic caching by developing and evaluating *Crystal*. Its long-term knowledge builds on semantic data regions based on the workload history. Unlike previous research on semantic caching, *Crystal* embraces data overlap and considers overlapping filters during knapsack computation. Our experiments showed that *Crystal* has superior cache utilization, quick adaptability, and significantly enhances query performance.

JSON tiles enables fast analytics on semi-structured data. Because developers often prefer the simplicity of semi-structured formats, a large amount of data stored in cloud object storage originated from JSON APIs. However, analytical query processing of JSON documents is slow due to access overhead. To overcome these issues, we introduced *JSON tiles*. In contrast to previous work, *JSON tiles* enhances robustness, performance, and query optimization by exploiting the structural information found in JSON documents. By materializing the frequent columns of a small collection of documents (a tile), *JSON tiles* naturally adapts to changes in the incoming data. Reordering of documents between tiles provides strong robustness guarantees. Compared to other industry-grade systems, *Umbra* with *JSON tiles* achieves at least one order of magnitude better query latencies for analytical workloads on semi-structured data.

Memory allocation impacts query processing. As the amount of stored and processed data continues to increase, efficient and fair memory allocation becomes more important. To help understand the performance implications of memory allocation, we presented the first experimental study of the impact of memory allocation for analytics. Interestingly, the five tested memory allocators demonstrated significant differences in terms of memory fairness, efficiency, scalability, and query performance. Our research revealed that jemalloc performed superior in most categories, which led to a switch to jemalloc as the default allocator for *Umbra*.

Facilitating cost-effective and efficient cloud-based data processing. This thesis presented a collection of solutions to pressing issues of analytical query processing in public clouds. By combining these ideas, database systems can become more cost-effective and achieve higher performance. Balancing local disks and remote cloud object storage is crucial for inexpensive and high-performing systems. Users demand fast analytical processing of heterogeneous data, particularly the processing of JSON documents, a pillar of big data and a state-of-the-art API format. To summarize, the architectural paradigm shift in the cloud of separating storage and compute governed the solutions in this thesis. Our redesigned and newly developed components enable modern database systems to become cloud-aware and process analytical queries efficiently and economically.

Future Work

1 Tbit/s Ethernet on the horizon. Public cloud vendors introduced 100 Gbit/s Ethernet in 2018, and 400 Gbit/s will be available soon. Recent research indicates that the next breakthrough in networking will be 1 Tbit/s Ethernet [Cai+22]. These high-bandwidth scenarios make cloud object storage even more appealing. However, reducing the CPU usage of retrieval is necessary to manage these fast networks efficiently. One solution is upgrading the default path MTU between instances and cloud object storage. Traditional protocols such as HTTP over TCP demand CPU resources by the number of processed packets, almost independent of the size. Increasing the size of network packets, for example by employing jumbo packets, can help to efficiently manage high-speed networks even on medium-sized servers. Faster networks may require adjusting the retrieval pipeline in database systems, for example, increasing the object retrieval concurrency.

Disaggregating memory is the next leap in elasticity. In the future, cloud vendors may provide not only disaggregated storage but also disaggregated memory [Kor+22; Wan+22; Zha+20; Zha+21]. CXL, a newly developed standard, appears to be a promising solution to extend byte-addressable memory during query processing. While this idea is promising and has already sparked research interest, hardware for seamless memory upgrades is not yet available. Provisioning additional memory on demand and the different access latencies between local and remote memory further complicates memory allocation. These new allocation patterns might require a redesign of how memory is allocated and when it is freed. In contrast to today's memory allocators, monetary cost can serve as a valuable metric for memory allocation.

Opportunities in control plane management and distributed processing. With the advancements achieved in this thesis, modern database systems can now leverage public cloud infrastructure for efficient analytical query processing. However, most commercial analytics offerings consist of multiple instances. Managing instances and scheduling queries on different instances is challenging. This is referred to as the control plane, which must be carefully designed to deliver a cost-effective, multi-instance analytics solution. When data cannot be managed on a single instance, cloud offerings typically distribute query processing. These topics are beyond the scope of this thesis but raise exciting research challenges that can build upon the presented solutions in this dissertation.

PAPER P1

Exploiting Cloud Object Storage for High-Performance Analytics

Synopsis. The disaggregation of storage and compute is an essential design principle in the cloud, and most cloud-native database systems rely on cloud object storage as the ground-truth storage solution. Cloud object storage, such as AWS S3, IBM COS, and GCP Storage, is inexpensive, scalable, and durable, making it the perfect fit for analytical query engines. Traditionally, the major challenge with cloud object storage was the limited network bandwidth between instances and storage. Due to recent advances in networking technology, cloud vendors offer affordable instances with 100 Gbit/s Ethernet or more.

This paper addresses the three key challenges associated with efficient analytics on cloud object stores. (i) Achieving complete network bandwidth on network-optimized instances is challenging because a large number of simultaneously outstanding requests are required. Therefore, efficient network retrieval must be tightly integrated with the database system. (ii) Network I/O incurs greater CPU overhead than local disk accesses, decreasing the number of cores available for concurrent analytics. Thus, reducing the CPU usage of network retrieval is essential. (iii) Many cloud database systems offer the flexibility to run on different cloud vendors. Yet, each cloud vendor provides its own network library, leading to increased complexity when integrating multiple libraries.

To mitigate these challenges, this paper presents a blueprint for performing efficient analytics on data residing in disaggregated cloud object stores. It conducts an analysis of cloud object stores from multiple vendors to establish retrieval configurations optimized for cost and performance. The results of the study are used to develop a download manager, called *AnyBlob*, designed for large data analytics. *AnyBlob* significantly reduces CPU resource consumption and can retrieve data from multiple cloud vendors. By integrating high-bandwidth object retrieval seamlessly with the database engine's scan operator, the database system *Umbra*, deployed on a single instance, achieves performance similar to large configurations of state-of-the-art cloud database systems.

Contributions. Dominik Durner contributed substantially to the content of the paper, in particular concerning the development of the proposed ideas, the implementation of the system, the evaluation, and authoring substantial parts of the paper.

Reference. Dominik Durner, Viktor Leis, and Thomas Neumann. "Exploiting Cloud Object Storage for High-Performance Analytics". In: *PVLDB* 16.11 (2023), pp. 2769–2782

DOI. `https://doi.org/10.14778/3611479.3611486`

Exploiting Cloud Object Storage for High-Performance Analytics

Dominik Durner
Technische Universität München
dominik.durner@tum.de

Viktor Leis
Technische Universität München
viktor.leis@tum.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Elasticity of compute and storage is crucial for analytical cloud database systems. All cloud vendors provide disaggregated object stores, which can be used as storage backend for analytical query engines. Until recently, local storage was unavoidable to process large tables efficiently due to the bandwidth limitations of the network infrastructure in public clouds. However, the gap between remote network and local NVMe bandwidth is closing, making cloud storage more attractive. This paper presents a blueprint for performing efficient analytics directly on cloud object stores. We derive cost- and performance-optimal retrieval configurations for cloud object stores with the first in-depth study of this foundational service in the context of analytical query processing. For achieving high retrieval performance, we present *AnyBlob*, a novel download manager for query engines that optimizes throughput while minimizing CPU usage. We discuss the integration of high-performance data retrieval in query engines and demonstrate it by incorporating *AnyBlob* in our database system *Umbra*. Our experiments show that even without caching, *Umbra* with integrated *AnyBlob* achieves similar performance to state-of-the-art cloud data warehouses that cache data on local SSDs while improving resource elasticity.

PVLDB Reference Format:
Dominik Durner, Viktor Leis, and Thomas Neumann. Exploiting Cloud Object Storage for High-Performance Analytics. PVLDB, 16(11): 2769 - 2782, 2023.
doi:10.14778/3611479.3611486

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/durner/AnyBlob.

1 INTRODUCTION

Data warehousing moves to the cloud. Estimates show that the revenue of cloud database systems has reached that of on-premise systems in 2021 [1] – and by VLDB 2023, the cloud market share will presumably be significantly higher. A major part of this change is the shift of data warehousing and analytical query processing to the cloud. The main drivers behind that are elasticity and the flexibility to provision storage and compute separately and on demand.
Cloud object stores. Cloud object stores such as AWS S3, IBM COS, and GCP Storage enable separating compute from storage in a cost-effective (e.g., ~23\$/TiB per month) way [13]. They also provide strong durability guarantees (e.g., 11 9's per year for S3 [4]), practically unlimited capacity, and scalable access bandwidth. These properties make disaggregated cloud object storage a natural fit for analytical database systems. In future data centers, database systems may even run on hardware that separates memory and compute. There, disaggregated storage is crucial to provide durability [83, 91].
High-bandwidth networks. Until recently, the major issue of cloud object storage for analytics was the limited network bandwidth between instances and storage. In 2018, AWS introduced instances with 100 Gbit/s ($\approx$12 GB/s) networking – resulting in a fourfold increase in per-instance bandwidth [22, 26]. In contrast to Infiniband, 100 Gbit/s Ethernet has not only become widely-available but also affordable[1]. This effectively closes the gap between remote network and local NVMe bandwidth[2] and makes relying more on cloud storage more attractive for bandwidth-dominated workloads.
Cloud storage analytics. Most cloud-native data warehouse systems, such as Snowflake [33, 82], Databricks [25], and AWS Redshift [19], use cloud object storage as their ground-truth data source. Although the bandwidth gap between local storage and network is closing, most research focuses on caching to avoid fetching data from remote storage [37, 46, 85, 89]. Early research investigates object storage for transactional database systems but limits its focus on OLTP [27]. Surprisingly, no empirical study for general-purpose analytics (OLAP) on cloud object stores has been conducted.
Challenge 1: Achieving instance bandwidth. Because the latency of each object request is high, saturating high-bandwidth networks requires many concurrently outstanding requests. Therefore, a careful network integration into the DBMS is crucial to achieve the complete bandwidth available on network-optimized instances.
Challenge 2: Network CPU overhead. In contrast to fetching data from local disks, network retrieval has higher CPU overhead. Query engines, however, also contend for computation resources to simultaneously analyze large sets of data. Consequently, reducing the CPU footprint of network retrieval is essential.
Challenge 3: Multi-cloud support. Many cloud database systems are able to run in different clouds – allowing the user to choose the vendor of their liking. In contrast to the desire for multi-cloud systems, each cloud vendor provides its own networking library. Thus, multiple libraries need to be integrated, which increases complexity.
Approach. In this paper, we present a blueprint for performing efficient analytics directly on data residing in disaggregated cloud object stores. We studied the cloud object stores of different vendors to derive cost- and performance-optimal retrieval configurations. To reduce resource utilization for network retrieval, we developed a download manager that is able to fetch data from multiple cloud vendors. We seamlessly integrate high-bandwidth object retrieval with the database engine's scan operator. Our DBMS Umbra, equipped

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611486

[1]Comparing the on-demand prices of c5n.18xlarge (100 Gbit/s) and c5.18xlarge (25 Gbit/s) while taking c5n's larger DRAM into account (~30% more DRAM), we find that adding 100 Gbit/s networking increases cost by only 22%.
[2]Consider i3en.24xlarge, the AWS instance with the fastest local NVMe bandwidth. Its local read bandwidth is 16 GB/s, while its full-duplex network bandwidth is 12 GB/s.

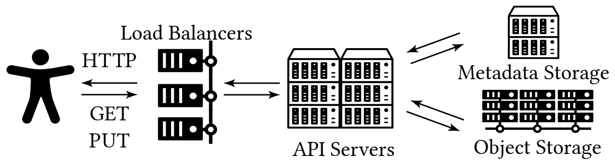


Figure 1: Schematic architecture of AWS S3.

with our download manager and caching disabled, achieves similar performance on a single instance as large configurations of state-of-the-art cloud database systems that cache data on local SSDs. Our fast and low overhead networking integration facilitates switching instances without performance cliffs, improving elasticity. As switching comes without performance cliffs, our approach is able to better utilize spot instances, available at huge discounts.

Contribution 1: Experimental study of cloud object stores. To achieve high-bandwidth data processing, we first study the properties of cloud object stores. In Section 2, we explain the design of disaggregated storage, discuss the cost structure, and then provide detailed experiments on the latency and throughput of different object stores. We define an optimal request size range that minimizes cost while maximizing throughput. Our concurrency analysis helps to schedule enough requests to meet the throughput goal (i.e., instance bandwidth). Our in-depth experimental study of this foundational cloud service helps to exploit disaggregated storage for analytical query processing.

Contribution 2: AnyBlob, a low overhead multi-cloud library. With the insights gained from our characterization of object stores, we developed *AnyBlob*, an open-source, multi-cloud download manager for object stores that is optimized for large data analytics [36]. *AnyBlob*, described in Section 3, achieves the same throughput as the libraries provided by the cloud vendors while reducing CPU resource consumption significantly. CPU resource utilization is vital to process data concurrently. In contrast to existing solutions, our approach does not need to start new threads for parallel requests because it uses `io_uring` [21], which facilitates asynchronous system calls. To saturate the network bandwidth, our analysis shows that hundreds of requests have to be outstanding simultaneously. Our solution helps to reduce thread scheduling overhead and allows seamless integration into database query engines.

Contribution 3: Blueprint for retrieval integration. Tight integration of the download manager into the database engine enables efficient analytics on disaggregated storage. We present a blueprint to incorporate *AnyBlob* into database engines in Section 4. By carefully designing the scan operator and developing an object retrieval scheduler, we can seamlessly interleave the downloading of objects with the analytical processing.

2 CLOUD STORAGE CHARACTERISTICS

Methodology. In order to design an efficient analytics engine based on cloud object storage, we need to understand its basic characteristics. We start with an analysis on the performance characteristics and cost of disaggregated object stores and compute instances. To gain insights into the storage architecture, we perform various micro-experiments on AWS S3 and two other cloud providers to understand latency and throughput limitations. A study on AWS shows that instances are able to achieve high network throughput to S3 [80]. With the best practices in mind [10], we conduct this in-depth experimental study that helps exploiting cloud storage for analytical query processing. Unless otherwise specified, we use our *AnyBlob* library as the retrieval manager, presented in Section 3.

Table 1: Cloud storage cost by cloud vendor for zone-redundant replication (default; multiple AZs within region).

Cloud Provider (cheapest region)	Storage ($ / TiB / month)	GET ($ / 1 M)	PUT ($ / 1 M)
AWS (us-east-2) [13]	23.55	0.40	5.00
GCP (us-east-1) [39]	20.48	0.40	5.00
IBM (us-east) [45]	23.55	0.42	5.20
Azure (East US 2) [60]	23.55	0.40	6.25
OCI (us-ashburn-1) [64]	26.11	0.34	0.34

2.1 Object Storage Architecture

Overview. All major cloud vendors provide disaggregated storage solutions such as AWS S3, Azure Blob, IBM COS, OCI Object Storage, and GCP Storage. Data is stored in immutable blocks called *objects*. These objects are distributed and replicated across several storage servers for availability and durability. After resolving the domain name of the cloud object store, the user requests an object from a storage server which then sends the data. All major cloud providers use a similar API that transfers data via HTTP (TCP).

Architecture of S3. The architecture of S3 is depicted in Figure 1 [32]. AWS S3 defines *prefixes* that are similar to unique file paths in operating systems. Objects are similar to files and all levels above objects are similar to directories. Data is stored in *buckets* that resemble hard drive partitions in our analogy. According to AWS, S3 partitions all prefixes to scale to thousands of requests per second [32]. A prefix can range from covering a bucket down to individual objects. With an update in 2020, S3 is now a strongly consistent system [23]. Other providers were already strongly consistent. S3 replicates the data to at least 3 different availability zones (AZs). A geographic region consists of AZs that are separated data centers for increasing availability and durability [75].

Bandwidth limits. Data access performance is characterized by the network connection of the instance, the network connection of the cloud storage, and the network itself. At AWS, general-purpose instances achieve 100 Gbit/s and more to the object stores [3, 5].

2.2 Object Storage Cost

Cost structure. All major cloud vendors structure their object storage pricing similarly. They categorize expenses as storage cost, data retrieval and data modification cost (API cost), and inter-region network transfer cost. Cloud providers operate object stores on the level of a region (e.g., eu-central-1). When accessing data within one region, only API costs are charged because intra-region traffic is free to the object store. On the other hand, AWS inter-region data transfer, for example, from the US east to Europe costs 0.02$/GB.

Size-independent retrieval cost. Table 1 shows that the pricing of cloud providers is similar for zone-redundant replication (default), which provides high durability and optimal retrieval performance.

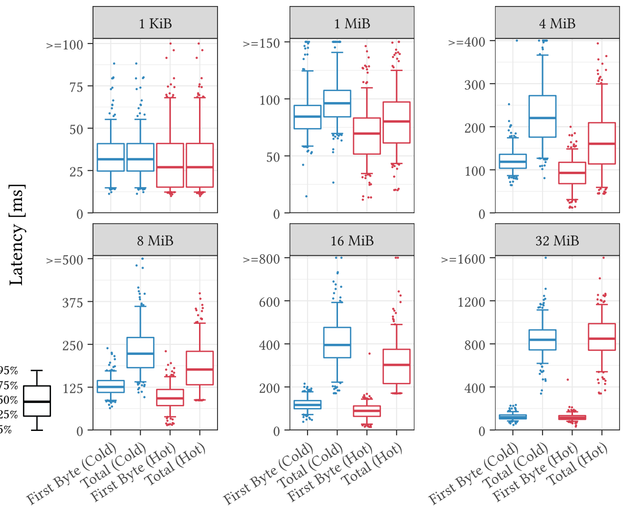


Figure 2: First byte and total latency for different requests sizes on hot and cold objects (AWS, eu-central-1, c5n.large).

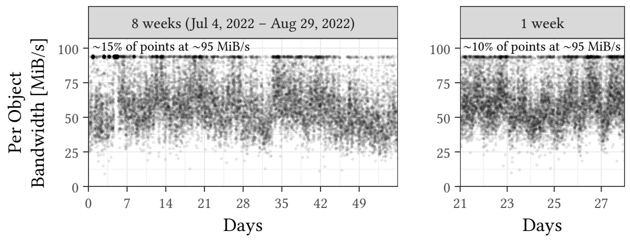


Figure 3: S3 bandwidth over 8 weeks (AWS, eu-central-1).

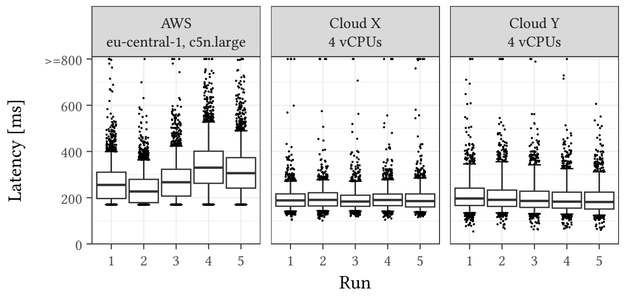


Figure 4: Total latency distribution of different object stores over multiple runs on sparsely accessed data (12h interval).

Surprisingly, retrieval cost in the same region depends only on the number of requests sent to the cloud object store, and does not depend on the object size. Downloading a 1 KiB object costs the same as a 1 TiB object, as long as only one HTTP request is issued.

Cloud storage alternatives. Other storage solutions are not as elastic as disaggregated storage and are often more expensive. For example, AWS Elastic Block Storage (EBS) (gp2 SSD) costs 102.4$/TiB compared to 23.2$/TiB per month. HDD storage pricing is comparable to S3, but bandwidth is very limited. Although EBS is elastic in its size, it can only be attached to a single node. Instance-based SSD storage is also expensive. For example, the price difference between c5.18xlarge and c5d.18xlarge is 0.396$/h and yields in 1.8 TB NVMe SSD. There, instance storage costs 158.4$/TB per month, which is 7× more expensive. Another example for instance-based storage is the largest HDD cluster instance d3en.12xlarge. This instance features 24 HDDs with 14 TB storage each at a price of 13.5$/TB per month. Although this seems cheaper initially, such an instance cannot provide S3's durability guarantees (11 9's). The parallelism of disaggregated storage enables higher throughput than local storage devices, which we will discuss in Section 2.8.

> **Finding 1:** Cloud object storage provides the best durability guarantees while being the cheapest storage option.

2.3 Latency

Different request sizes. Disaggregated storage incurs higher latency than SSD-based storage solutions. We examine the latency distribution for different request sizes to understand storage latency. We distinguish between total duration and latency until the first byte is retrieved. The results of using only a single request at a time are depicted in Figure 2. We differentiate between the first and 20th consecutive iteration to simulate hot accesses. Our experiment shows that first byte latency often dominates the overall runtime for small sizes. First byte and total duration are similar for small request sizes. This highlights that round-trip latency limits the overall throughput. For sufficiently large requests, bandwidth is the limiting factor. From 8 to 16 MiB, we see minor improvements but the duration already rises by ∼1.9× while object size doubles. Increasing the size from 16 to 32 MiB results in doubling the retrieval duration. Thus, the bandwidth limit is reached, and further increasing the size does not benefit the retrieval performance. When data is hot, first byte and total latency are generally reduced.

Noisy neighbors. Cloud-based storage solutions are shared between customers, resulting in less predictable latency. We continuously retrieve a single object from a set of objects with one request to analyze trends in access performance. We generate random 16 MiB objects since increasing the size does not lead to a lower latency per byte. Figure 3 shows the bandwidth for accessing an object (bytes divided by duration) over a period of 8 weeks. Object bandwidth has a high variance ranging from ∼25 to 95 MiB/s, with a considerable number of data points being at the maximum (15%). The median performance stabilizes at 55-60 MiB/s. Weekly patterns in the data show that the bandwidth is influenced by the day of the week. Especially at the weekends (first day of the week is Monday), the performance is higher – most likely due to lower demand from other customers. When we zoom into one week, clear daily patterns are visible. The performance fluctuations between day and night indicate variations in network utilization during different times of the day. Surprisingly, no outlier lies above the large cluster at ∼95 MiB/s even though millions of objects were downloaded. This suggests that the per-request bandwidth is limited within S3 or that server-side caching effects are intentionally not passed on to users.

Latency variations between cloud vendors. In addition to using AWS, we also examine latency characteristics of two other cloud

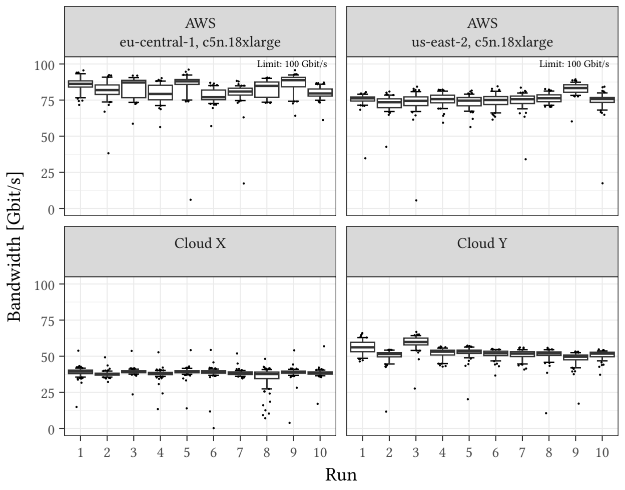


Figure 5: Throughput distribution of different object stores over multiple runs on sparsely accessed data (12h interval).

providers. The experiment, plotted in Figure 4, accesses randomly generated 16 MiB objects. After each run, the bucket is not accessed until the next run. The interval between executions is (at least) 12 hours to reduce caching effects. AWS S3 has the highest overall latency for individual objects. The other two providers have similar average latencies, but Cloud Y has more variance. Latency between different executions is fairly stable across all cloud providers. As mentioned, S3 has a minimum latency with no outliers below it, which suggests a restricted per-request maximum bandwidth. In contrast to AWS, outliers in the low latency spectrum indicate that the other vendors do not hide caching effects.

2.4 Throughput

Importance of throughput. Aside from latency, we also show insights on the throughput of accessing object stores. For analytics, the most important factor is the combined throughput since OLAP requires large amounts of data to be processed. Thus, the first byte latency is less important for bandwidth-dominated workloads.
Cloud storage throughput similar to instance bandwidth. Similar to our previous latency experiment, we access randomly generated 16 MiB objects. One request retrieves one complete object. In this experiment, we maximize the throughput available on each cloud provider with a single instance. We schedule up to 256 simultaneous requests using many threads to maximize throughput. Further increasing requests did not lead to higher throughput. Section 2.8 discusses the optimal number of requests. We use instances that achieve up to 100 Gbit/s (or the cloud's maximum bandwidth) and have similar on-demand pricing. Figure 5 shows the throughput experiment with (at least) 12 hours between different runs to reduce caching effects. Each throughput data point is calculated as an aggregate of all downloaded objects over a 1-second window. The results show that we achieve a median bandwidth of at least 75 Gbit/s for AWS. Most runs have a median bandwidth between 80 and 90 Gbit/s in eu-central-1, close to the maximum instance bandwidth. At Cloud X, we observe a bandwidth limit of ~40 Gbit/s

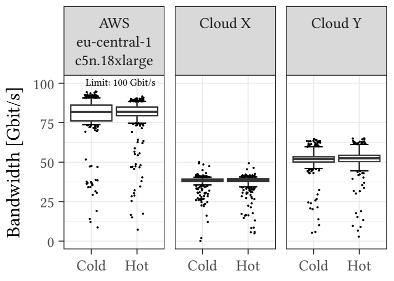


Figure 6: Throughput comparison of cold and hot runs.

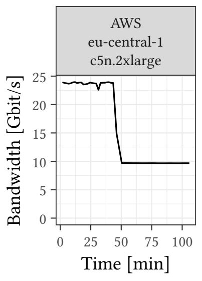


Figure 7: Instance burst bandwidth.

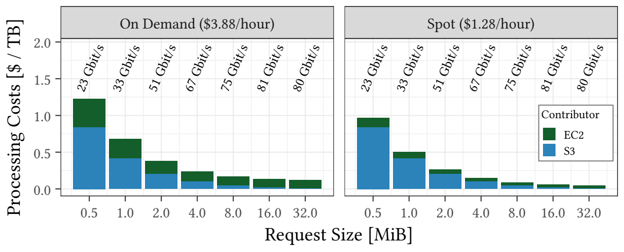


Figure 8: Cost vs. throughput of different request sizes (AWS, eu-central-1, c5n.18xlarge).

and almost no fluctuations. Cloud Y achieves a median bandwidth of 50 Gbit/s to its object store, but we notice higher variance.
Different regions have slightly different performance. Throughput is similar for the two tested regions of AWS; however, one region performs slightly better. The difference between the two regions does not vary much between iterations.
High bandwidth is achievable for cold objects. In Figure 6, we investigate the throughput differences between the first and the 20th consecutive execution. The access frequency spike of the same objects does not result in vastly different execution times.
Small instances allow bursting. In the AWS instance specifications, the network bandwidth of smaller instances is often denoted with an up-to bandwidth limit. Instances achieve the baseline bandwidth (relative to the number of CPUs) in the steady state after utilizing all burst credits [14]. Figure 7 shows that the instance falls back to the baseline throughput after bursting for ~45 min.

> **Finding 2:** Object retrieval can reach network bandwidth.

2.5 Optimal Request Size

Request size implications. An important design decision is the size of requests. Requests can either be full objects or byte ranges within objects. The most crucial factors are performance and request cost. Since cloud providers charge by the number of requests, larger requests result in lower cost for the same overall data size. On the other hand, the size should be as small as possible so that small tables also benefit from parallel downloads. Our experiments in Section 2.3 demonstrate that performance does not improve beyond the bandwidth limit for a single request.

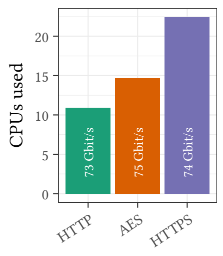


Figure 9: Impact of encryption on CPU usage.

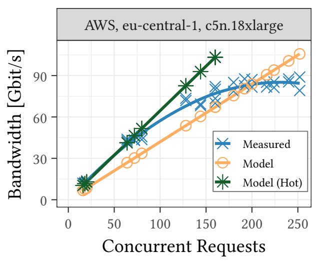


Figure 10: Request modeling for reaching throughput goal.

Cost-throughput optimal requests. In Figure 8, we show the cost of retrieving data from S3 with different request sizes. The achieved throughput with hundreds of simultaneous requests and many threads is denoted above the bars. Each request size class contains randomly generated objects. We distinguish between compute instance cost (c5n.18xlarge) and storage retrieval cost. Storage cost dominates the total cost for small objects. Computational cost is the most significant contributor to requests in the ~10 MiB range. This applies to instances at on-demand prices and spot instances, which come at a huge discount (we calculate with 60%). Because the throughput plateaus in the same range of request sizes, we classify request sizes of 8 - 16 MiB as cost-throughput optimal for OLAP.

Finding 3: Sizes of 8 - 16 MiB are cost-throughput optimal.

2.6 Encryption

CPU consumption of encryption. All experiments so far use an unsecured connection to S3 (HTTP), but S3 also supports encrypted connections through HTTPS. We measure the CPU overhead of different encryption strategies while reaching the same throughput in Figure 9. HTTPS requires more than 2× CPU resources of HTTP, but AES end-to-end encryption only increases CPU usage by ~30%.
Encryption-at-rest superior to HTTPS. At AWS, all traffic between regions and even all traffic between AZs is automatically encrypted by the network infrastructure. Thus, all traffic leaving an AWS physical location is automatically secured [8]. Within a location, no other user is able to intercept traffic between an EC2 instance and the S3 gateway due to the isolation of VPCs, making HTTPS superfluous. However, encryption-at-rest is required to ensure full data encryption outside the instance (e.g., at S3).

2.7 Tail Latency & Request Hedging

Hedging against slow responses. Missing or slow responses from storage servers are a challenge for users of cloud object stores. In our latency experiments, we see requests that have a considerable tail latency. Some requests get lost without any notice. To mitigate these issues, cloud vendors suggest restarting unresponsive requests, known as request hedging [10, 34]. For example, the typical 16 MiB request duration is below 600ms for AWS. However, less than 5% of objects are not downloaded after 600ms. Missing responses can also be found by checking the first byte latency. Similarly to the duration, less than 5% have a first byte latency above 200ms. Hedging these requests does not introduce significant cost overhead.

2.8 Model for Cloud Storage Retrieval

Concurrency analysis. During our analysis, we saw that the bandwidth of individual requests is similar to accessing data on an HDD. To saturate network bandwidth, many simultaneous requests are required. Requests in the range of 8 - 16 MiB are cost-effective for analytical workloads. We design a model to predict the number of requests needed to reach a given throughput goal:

$$\text{requests} = \text{throughput} \cdot \frac{\text{baseLatency} + \text{size} \cdot \text{dataLatency}}{\text{size}}$$

For sufficiently large request sizes at S3, we calculate the median base latency as ~30 ms and the median data latency as ~20 ms/MiB The base latency is computed from the 1 KiB experiment in Figure 2, the average latency of 16 MiB minus the base latency defines the median data latency. Figure 4 shows that the median data latency of Cloud X and Cloud Y is lower (12–15 ms/MiB). For S3, the optimal request concurrency for saturating 100 Gibt/s instances is ~200–250. Figure 10 evaluates the model with the previous data latency and the latency representing the 25th percentile (hot). The measurements are between both models until the bandwidth limit is reached.
Storage medium. An access latency in the tens of ms and a per-object bandwidth of ~50 MiB/s strongly suggest that cloud object stores are based on HDDs. This implies that reading from S3 with ~80 Gbit/s is accessing on the order of 100 HDDs simultaneously.

Finding 4: Saturating high-bandwidth networks requires hundreds of outstanding requests to the cloud object store.

3 ANYBLOB

Unified interface with smaller CPU footprint. Different cloud providers have their own download libraries with different APIs and performance characteristics [7, 40, 44, 61, 65]. To offer a unified interface, we present a general-purpose and open-source object download manager called *AnyBlob* [36]. In addition to transparently supporting multiple clouds, our `io_uring`-based download manager requires fewer CPU resources than the cloud-vendor-provided ones. Resource usage is vital as our download threads run in parallel with the query engine working on the retrieved data. Existing download libraries start new threads for each parallel request. For example, the S3 download manager of the AWS SDK executes one request per thread using the open-source HTTP library `curl`. In contrast to spinning up threads for individual requests, *AnyBlob* uses asynchronous requests, which allows us to schedule fewer threads. Because hundreds of requests must be outstanding simultaneously in high-bandwidth networks, a one-to-one thread mapping would result in thread oversubscription. This results in many context switches, which negatively impacts performance and CPU utilization.

3.1 AnyBlob Design

Multiple requests per thread. *AnyBlob* uses `io_uring` to manage multiple connections per thread asynchronously [31]. With this model, the system does not have to oversubscribe threads which would incur additional scheduling cost. In the following, we discuss the three major components of *AnyBlob*. The components and their relationship are shown in Figure 11.
io_uring - low-overhead system call interface. `io_uring` (available since Linux kernel 5.1) provides a generic kernel interface for

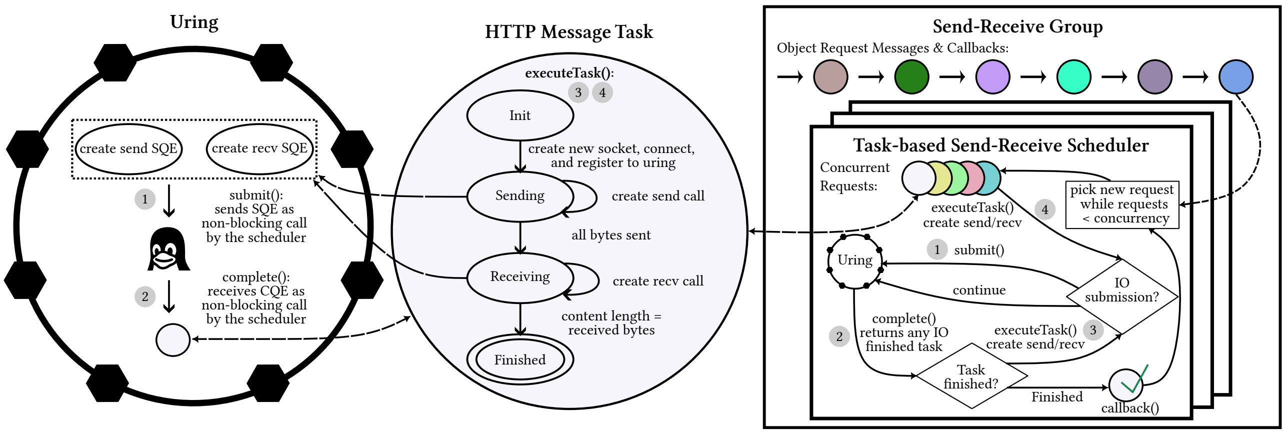

Uring
create send SQE
create recv SQE
1
submit(): sends SQE as non-blocking call by the scheduler
2
complete(): receives CQE as non-blocking call by the scheduler
HTTP Message Task
executeTask():
3 4
Init
create new socket, connect, and register to uring
Sending
create send call
all bytes sent
Receiving
create recv call
content length = received bytes
Finished
Send-Receive Group
Object Request Messages & Callbacks:
Task-based Send-Receive Scheduler
Concurrent Requests:
executeTask() create send/recv
4
pick new request while requests < concurrency
Uring
1 submit()
continue
IO submission?
complete() returns any IO finished task
2
executeTask() create send/recv
3
Task finished?
Finished
callback()

Figure 11: AnyBlob uses state-machine based message tasks that are asynchronously processed with the help of `io_uring`.

storage and network tasks. It builds on two lock-free ring buffers, the submission and completion queues, that are shared between user and kernel space. The user inserts new submission queue entries (SQE), such as receive (recv) and send operations, into the submission queue. Inserting into the queue does not require any syscalls. To notify the kernel of new entries in the submission queue, the `io_uring_enter` system call processes the entries on the kernel side in a non-blocking fashion until the request is transmitted to the network or storage device. This device uses interrupt requests (IRQs) to notify the kernel of finished operations. The request is then processed during the interrupt and placed on the completion queue. To check if a request was successful, the user periodically peeks for available completion queue entries (CQE). `io_uring` was found to be highly efficient for storage applications [35, 41, 52, 55, 68], but is less studied for networking tasks [28]. Didona et al. suggest to study `io_uring` for networking in more depth [35].

State-machine-based messages. *AnyBlob* uses a state machine for each request. The message information (address, port, and raw data) combined with a state machine is denoted as a *Message Task*. Optionally, a receive buffer can be attached that avoids additional data copies since the kernel transfers data directly from the network device to our desired location. Cloud object stores use HTTP messages to transfer data. We implement the different phases of an HTTP request within the state machine. On successfully completing a phase, we transition to the next phase until the object is fully fetched. The state machine enables asynchronous and multiplexed messages with a single thread. Several send and recv system calls are required during transfer until the object is downloaded. After each system call, we suspend the execution of this message until we are notified about the successful syscall. Afterward, we reevaluate the state machine until a final state is reached.

Asynchronous system calls. Our asynchronous handling of send and recv system calls in the *Message Task* is facilitated by `io_uring`. Instead of directly scheduling the system call and waiting for the result, we insert the operation into the submission queue of the uring. Each SQE has a user-defined member that allows passing information to later identify its origin *Message Task*. System calls are processed only when the submission queue is submitted to the kernel (1). This submission process is non-blocking, allowing the executing thread to work on other requests while the transfer is handled by the network device. The uring is periodically checked for available completion queue entries (CQE) (2). When a CQE is available, a system call has been processed. With the retrieved information, we can evaluate the next *Message Task* step.

Task-based send-receive scheduler. With `io_uring`-based sockets and *Message Tasks*, we develop a task-based send-receive scheduler. The task scheduler uses one thread that continuously executes (1) – (3) as an event loop. This event loop coordinates the execution of the steps of *Message Tasks* (3) and processes completion entries (2). Furthermore, new object requests are scheduled as new *Message Tasks* (4). To optimize single-threaded throughput, a task scheduler works concurrently on multiple *Message Tasks.* Multiple *Message Tasks'* send and recv system calls can be batched before submitting the submission queue to reduce system call overhead (1). In multi-threaded environments, it is beneficial to reduce system calls as parts of them are protected by kernel locks. When a *Message Task* is finished, it invokes a callback to notify the requester.

Send-receive groups. Although a single task-based send-receive scheduler has high throughput (multiple Gbit/s), it is not sufficient to satisfy network-optimized instances. Thus, multiple schedulers need to run simultaneously. For ease of use, a lock-free send-receive task group manages requests for multiple send-receive schedulers.

3.2 Authentication & Security

Transparent authentication. Although all cloud providers use a similar API to access objects, some details of signing requests and the authentication are different. *AnyBlob* implements operations to upload and download objects from multiple cloud storage providers. We implement a custom signing process using the library `openssl` to maintain high throughput with as few cores as possible [20]. For users of *AnyBlob*, it is transparent which provider is chosen, as the interaction with the library remains unchanged. For AWS, we support the automatic short-term key metadata service [11].

AnyBlob enables encryption-at-rest. *AnyBlob* supports the user application to use encryption-at-rest by providing easy-to-use, in-place, and fast encryption and decryption functions for AES. Further, *AnyBlob* allows the usage of HTTPS for requests. However, we discourage this in controlled environments, such as AWS EC2 connected to AWS S3, due to high CPU overhead. HTTPS is useful for authentication if data is sent outside the controlled environment, e.g., from your computer to S3. In contrast to the high overhead for HTTPS, encryption-at-rest can be used with only moderate overhead. As shown in Section 2.6, this client-side encryption provides superior encryption against third parties, e.g., cloud providers.

3.3 Domain Name Resolver Strategies

Resolution overhead. In analytical scenarios, many requests are scheduled to the cloud object storage. Section 2.1 highlights that we can connect to different server endpoints. Resolving a domain name for each request adds considerable latency overhead due to additional round trips. Thus, it is essential to cache endpoint IPs.
Throughput-based resolver. Our default resolver stores statistics about requests to determine whether an endpoint is performing well. We cache multiple endpoint IPs and schedule requests to these cached IPs. If the throughput of an endpoint is worse than the performance of the other endpoints, we replace this endpoint. Thereby, we allow the load to balance across different endpoints.
MTU-based resolver. We found that the path maximum transmission unit (MTU) differs for S3 endpoints. In particular, the default MTU to hosts outside a VPC is typically 1500 bytes. Some S3 nodes, however, support Jumbo frames using an MTU of up to 9001 bytes [9]. Jumbo frames reduce CPU cost significantly because the per-packet kernel CPU overhead is amortized with larger packets.
MTU discovery. The S3 endpoints addressable with a higher path MTU use 8400 bytes as packet size. Our AWS resolver attempts to find hosts that provide good performance and use a higher path MTU. We ping the IP with a payload (> 1500 bytes) and set the DNF (do not fragment) flag to determine if a higher path MTU is available.

3.4 Performance Evaluation

Competitors. To demonstrate *AnyBlob*'s performance and CPU usage utilization, we experiment with different settings on AWS. We compare against two libraries provided by Amazon. They are both part of the official AWS C++ SDK (1.9.140). S3 is the traditional API that uses the library `curl` internally to retrieve objects. Similar concepts are applied by the download managers of other vendors' SDKs. S3Crt is a newer alternative S3 library released by AWS that uses a custom C network implementation (C++ API). With *AnyBlob*'s design, S3 Select can be implemented, but it would only support few types (JSON, CSV, Parquet) and no client-side encryption [15].
Cost-throughput Pareto-optimal retrieval. Figure 12 shows different settings for each tested download manager. Note that we plot performance and CPU utilization such that the optimal settings lie in the top-left corner of the Pareto curve. Within one download strategy, we highlight the points on their respective Pareto curve. *AnyBlob*, with our throughput-based resolver, always dominates the AWS-provided download managers. We achieve the same maximum throughput using only 0.7× the CPU resources of the best competitor. Given a fixed CPU budget, we get up to 1.5× performance. Our specialized AWS resolver achieves the same throughput but reduces CPU usage by an additional 10%. We validated *AnyBlob* on recently deployed Graviton instances (200 Gbit/s) [5] and observed greater CPU reduction while retrieving objects with up to 180 Gbit/s.

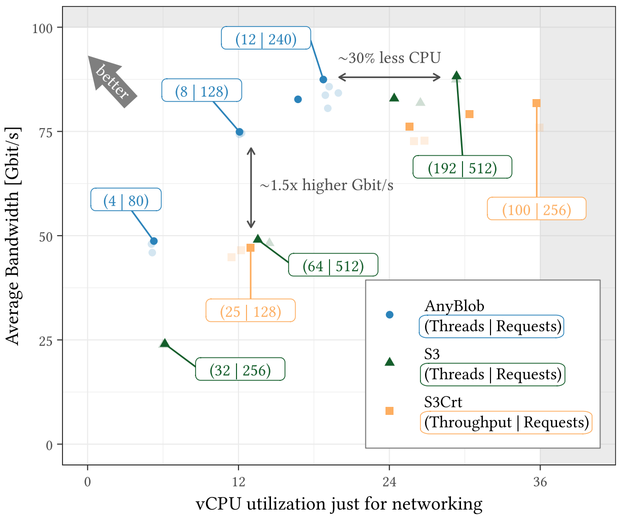


Figure 12: Throughput and CPU usage Pareto curves for AnyBlob, S3, and S3Crt (AWS, eu-central-1, c5n.18xlarge).

4 CLOUD STORAGE INTEGRATION

Query engine integration options. To unleash the full performance potential of disaggregated cloud storage, we have to carefully integrate the analytical query engine with the networking components. A naive approach would let each worker thread download its currently-needed data chunk synchronously. This way, each worker thread would schedule at most one request at a time, but the threads would be blocked most of the time – waiting for network I/O. A more common approach in database systems is the usage of asynchronous I/O. Our cloud storage retrieval approach builds upon this common I/O strategy. Database systems that use the AWS S3 SDK [7] also leverage asynchronous retrieval from cloud object storage. As discussed in Section 3, the AWS S3 SDK often results in oversubscription, which has not only a negative impact on performance but also other undesirable effects on database systems. For example, a huge download task with hundreds of threads could make the DBMS unresponsive to newly arriving queries since the DBMS has no control over the retrieval threads. Furthermore, the mix of downloading and processing threads is hard to balance, especially with this vast number of concurrently active threads.
Approach. In this section, we show how to integrate efficient object store retrieval into high-performance query engines. We rely on *AnyBlob* and the empirical results presented in Section 2 to saturate the available network bandwidth with low CPU resource consumption. A key challenge is how to balance query processing and downloading. Without enough retrieval threads, the network bandwidth limit can not be reached. On the other hand, if we use too few worker threads for computation-intensive queries, we lose the in-memory computation performance of our DBMS. We, therefore, propose a scheduling component to balance object store retrieval and query processing, allowing us to schedule threads effectively in terms of query performance and CPU usage. With this scheduler, we then develop an efficient table scan operator based on a cost-effective columnar storage format.

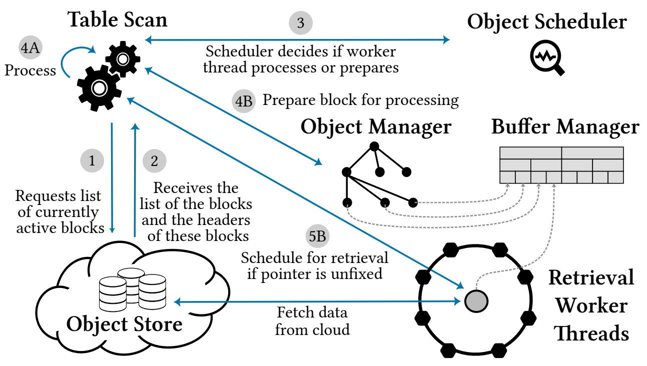


Table Scan
3
Object Scheduler
4A
Process
Scheduler decides if worker thread processes or prepares
4B Prepare block for processing
Object Manager
Buffer Manager
1
2
Requests list of currently active blocks
Receives the list of the blocks and the headers of these blocks
5B
Schedule for retrieval if pointer is unfixed
Retrieval Worker Threads
Object Store
Fetch data from cloud

Figure 13: DBMS design overview for efficient analytics with the flow of information between different components.

4.1 Database Engine Design

Tasks and scheduling of worker threads. The overall design of our cloud storage-optimized DBMS centers around the table scan operator. Like most database systems, our system Umbra uses a pool of worker threads to process queries in parallel. In our design, worker threads do not only perform (i) regular query processing, but can also (ii) prepare new object store requests or (iii) serve as network threads. Our object scheduler, which we present in Section 4.3, dynamically determines each worker's job (i-iii) depending on network bandwidth saturation and processing progress.

Task adaptivity. To overcome issues with long-running queries that block resources, many database systems use tasks to process queries. These tasks can either be suspended or run only for a small amount of time. Both concepts lead to a query engine that is able to adapt to changing workloads quickly. Regardless of the specific task system, our asynchronous retrieval integration only requires the mechanism to switch tasks of workers during query runtime.

Columnar format. The raw data is organized in a column-major relation format chunked in immutable blocks of columns. The metadata of a block, e.g., column types and offsets, are stored in the block header. The database schema information is also stored on cloud storage, which requires fetching at start-up.

Table metadata retrieval. In the following, we describe the flow of information during a table scan operation, illustrated in Figure 13. In steps 1 and 2, the scan operator first requests table metadata, i.e., the list of blocks. Afterward, all relevant block metadata is downloaded as a requirement to start the table scan's data retrieval.

Worker thread scheduling. After initializing the table scan, we dedicate multiple worker threads to this operation. Because partitioning worker threads into retrieval and processing threads is difficult and requires adaptations over the duration of the query, we implement an object scheduler to solve this problem. Step 3 shows that each scanning thread asks the scheduler which job to work on. If enough data is retrieved, the worker thread proceeds to process data, as demonstrated in 4A. Otherwise, we dedicate the thread to preparing blocks for retrieval. Since we only execute jobs for a short time, this decision can be quickly adapted.

Download preparation. To saturate the network bandwidth, it is important to continuously download with enough retrieval threads and many outstanding requests. In Step 4B, the preparation worker

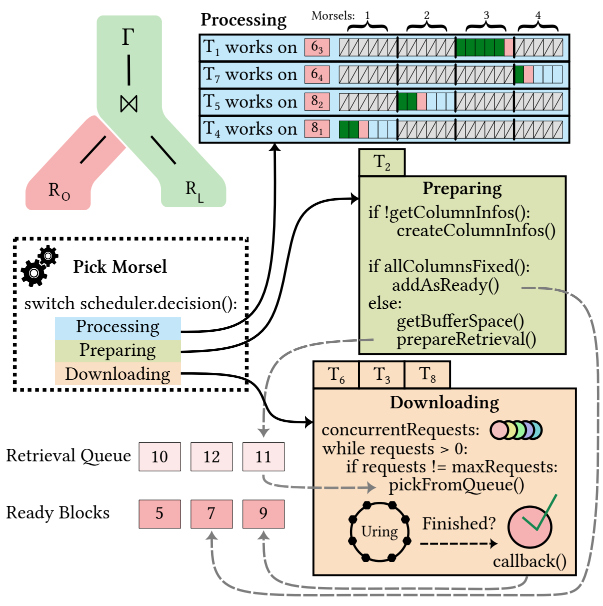


Figure 14: Table scan example with 8 threads.

creates new requests that allow the retrieval threads to execute their event loop without interruption. The object manager holds metadata of tables, blocks, and their column chunk data. The column chunk data is managed by our variable-sized buffer manager. If the data is not in memory, we create a new request and schedule it for retrieval, shown in 5B. Finally, retrieval threads fetch the data.

4.2 Table Scan Operator

Scan design preliminaries. We carefully integrate *AnyBlob* into our RDBMS Umbra, which compiles SQL to machine-code and supports efficient memory and buffer management [38, 48, 63]. Umbra uses worker threads to parallelize operators, such as table scans, and schedules as many worker threads as there are hardware threads available on the instance. If there is only one active query, all workers are used to process that specific query. Umbra's tasks consist of morsels which describe a small chunk of data of the task [53]. Worker threads are assigned to tasks and process morsels until the task is finished or the thread is assigned to a different task.

Morsel picking. After Umbra initializes the table scan, the worker threads start calling the `pickMorsel` method. This function assigns chunks of the task's data to worker threads. This is repeated after each morsel completion as long as the thread continues to work on this table scan task. The only difference in our approach is that our workers do not only need to process data but also prepare new blocks or retrieve blocks from storage servers. Our object scheduler, which we explain in Section 4.3, decides the job of a worker thread based on past processing and retrieval statistics. Note that similar to our `pickMorsel`, every task-based system has a method that determines the next task of a worker thread.

Worker jobs. If a thread is assigned to process data, a morsel is picked from the currently active block in `pickMorsel`. In contrast to the processing job, the other jobs (preparation and retrieval) do not pick a morsel for scanning. Instead, these jobs start routines

Algorithm 1: Scheduler: Adaptivity Computation

```
1 retrieveSpeed = statistics[epoch].retrievedBytes / statistics[epoch].elapsed
2 processSpeed = (workerThreads - currentRetriever) *
    statistics[epoch].processedBytes / statistics[epoch].processedTime
3 ratio = processSpeed / retrieveSpeed
4 requiredBandwidth = min(bandwidth, bandwidth * ratio)
5 requiredRetrieverThreads = min(maxRetrievers * ratio, maxRetrievers)
```

that are required to prepare or retrieve blocks. Regardless of the job, all workers return to `pickMorsel` to get a new job assigned after finishing their current work.

Scan example overview. Figure 14 shows the full table scan operation with multiple (8) active threads working on different jobs. In the example table scan, 4 threads are dedicated to processing data, 3 for data retrieval, and 1 for preparing new blocks.

Processing job. After receiving a morsel for processing, the thread scans and filters the data according to the semantics of the table scan. When all morsels of an active block (global or thread-local with stealing) are taken, the thread picks the morsel from a new, already retrieved block. In the example, each block is divided into 4 non-overlapping morsels. Each thread works on its unique morsel range.

Preparation job. With the already retrieved table metadata, threads prepare new blocks and register unknown blocks in the object manager. If the data of all columns currently resides in physical memory, the preparing thread marks the block as ready. Otherwise, the preparing thread gets free space from the buffer manager for each unfixed column. With the block metadata (column type, offset, and size), HTTP messages for fetching columns from cloud storage are created. After that, the block is queued for retrieval, where the data is downloaded.

Retrieval job. In the example, three threads are scheduled to act as *AnyBlob* retrieval threads. After finishing the download of a block's column chunk, a callback is invoked and marks this column as ready. Only if all columns have been retrieved, we mark the block as ready. Note that different retrieval threads may download column chunks from the same block concurrently. The worker finishes when *AnyBlob*'s request queue gets empty. Because threads always try to keep the queue at its maximum request length, unnecessary retrieval threads will eventually encounter an empty queue and stop downloading. These threads can then be reused to work on different jobs, such as processing or preparing new blocks. As long as enough requests are in the queue, the threads constantly retrieve data.

4.3 Object Scheduler

Balance of retrieval and processing performance. The main goal of the object scheduler is to strike a balance between processing and retrieval performance. It assigns different jobs to the available worker threads to achieve this balance. If the retrieval performance is lower than the scan performance, it increases the amount of retrieval and preparation threads. On the other hand, reducing the number of retrieval threads results in higher processing throughput. Note that the retrieval performance is limited by the network bandwidth, which the object scheduler considers.

Processing and retrieval estimations. The decision process requires performance statistics during retrieval and processing. Each processing thread tracks the execution time and the amount of data processed. The aggregated data allow us to compute the mean processing throughput per thread. For the network throughput, we aggregate the overall retrieved bytes during our current time epoch.

Balancing retrieval threads and requests. Sections 2.8 and 3.4 analyze how many concurrent requests are needed to achieve our throughput goal and the corresponding number of *AnyBlob* retrievers. We track the number of threads used for retrieval and limit it according to the instance bandwidth specification. By counting the number of outstanding requests (e.g., column chunks), we compute an upper bound on the outstanding network bandwidth. An outstanding request is a prepared HTTP request currently downloaded or awaiting retrieval. Because the number of threads and the outstanding requests limit the network bandwidth, our object scheduler always requires that the outstanding bandwidth is at least as high as the maximum bandwidth possible according to the current number of retrieval threads. Hence, it schedules enough preparation jobs to match the number of retrieval threads.

Performance adaptivity. The scheduler computes the global ratio between processing and retrieval to balance the retrieval and processing performance. This ratio is used to adapt the number of retrieval threads and the outstanding bandwidth. If processing is slower, fewer blocks are prepared, and fewer retrieval threads are scheduled. Some of the running retrieval threads will stop due to fewer outstanding requests. These threads are then scheduled as processing workers, increasing the global processing performance. Algorithm 1 shows these adaptivity computations.

Overpreparation. Because it is undesirable to stall retrieval threads due to unprepared columns, overpreparation is encouraged. Our scheduler ensures that up to 2× of the required bandwidth is outstanding and schedules preparation jobs accordingly.

Fast statistics aggregation. Lock-free atomic values for statistics and global counters provide fast object scheduler decisions. For every new scan request, we update the epoch to store representative statistics of the current workload.

4.4 Relation & Storage Format

Columnar format. To leverage the cost-throughput optimal download sizes, we require a column-major format that is chunked into different blocks. The database format is adapted from data blocks [51]. For each column chunk, we store min and max values in the metadata, enabling us to prune unnecessary blocks early. Our blocks use low-overhead byte-level encodings, e.g., frame-of-reference and dictionaries, to reduce storage requirements.

Tuple count in blocks. For cost-effective downloading, each column chunk of a block should have a desired size of 16 MiB. As query processing usually works on a block granularity, all columns within one block need to have the same number of tuples. However, this results in imperfect column chunk sizes due to different datatype sizes and our byte-level encoding scheme. The range per tuple in an encoded column is between 1 and 16 bytes, excluding the variable-sized columns. Because of this wide byte spread, we need to balance the sizes of the individual column chunks by optimizing the tuple count. During block construction, we adaptively compute mean tuple counts such that no encoded column falls below ~2 MiB to limit retrieval cost. Some fixed-sized and variable-sized columns may exceed 16 MiB, which is undesirable for retrieval. To avoid

large differences in download latency between columns, Umbra splits larger column chunks into multiple smaller range requests.
Zero user-space copies. Our implementation is tightly coupled with the buffer manager to reduce copies of data. The blocks of data are aligned to the page sizes of the buffer manager, but we reserve space for the HTTP header and the chunk size of the recv system call. By using the result data offset, we avoid user-space data copies.
Transparent paging. We extend the buffer manager with anonymous pages not backed by files to take advantage of the paging and in-memory buffer management features. If a new retrieval on the same page is necessary, we check if the page is still available. If not, we download the data again. With this unified and transparent buffer manager, we avoid retrieval and buffer space trade-offs.
Structure of metadata. Figure 15 shows the object structure in the cloud object storage. Within the database prefix, we store the schema information that contains all the necessary information to initialize the database. Each table has its own subprefix, which contains a list of headers, headers, and data blocks. Because header objects are also cost-throughput optimized, we store fewer header objects than blocks, as each header object contains multiple block headers. The data is organized for append-only storage, which mimics most analytical engines. Because objects can be replaced atomically in cloud storage, updating the list of headers creates consistent data snapshots. Versioning the metadata is common in cloud DBMSs to provide consistent views of the data. Apache Iceberg [17] and Data Lake [18] use an analogous technique. Iceberg's manifest files are similar to our list of headers and header objects [43].
Scan optimizations. Our implementation checks a header's min/-max values to avoid unnecessary downloads. A block is only scheduled for retrieval if all table scan restrictions match the min/max values within the block metadata. Before scanning the encoded data, the processing thread has to decode the data. We repeatably fill a small chunk with decoded data and process it. Umbra can either decode the data entirely or only decode tuples that satisfy the restrictions. Both approaches leverage vectorized SIMD instructions.

4.5 Encryption & Compression

Size reduction with strong compression. Although the bandwidth to external storage is high, modern engines might still wait on data arrival. With the encoding schemes presented in Section 4.4, the size of the columns is already reduced. Additional stronger compression allows for reducing them further. We use bit-packing for integer-encoded columns and apply LZ4 on the remaining ones.
Security due to encryption-at-rest. As already described in Section 2.6, encryption-at-rest does not only secure the traffic in transit but also stores the data inaccessible to third-parties. Before uploading a column, we use *AnyBlob* to encrypt the individual columns of a block. Although encryption with AES has a slight performance penalty, most real-world users prefer the gained security benefits.

5 EXPERIMENTAL EVALUATION

Setup. We extended our high-performance database system Umbra to support efficient analytics on disaggregated cloud object stores. All experiments are conducted at AWS in region *eu-central-1*. Unless otherwise noted, we use a single *c5n.18xlarge* (72 vCPUs / 36 cores, 192 GiB main memory, 100 Gbit/s network) instance with Ubuntu.

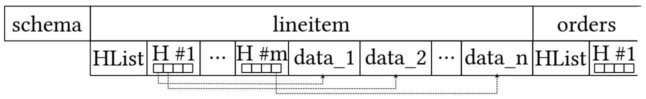


Figure 15: Object structure overview on S3 for TPC-H.

5.1 Data Retrieval Performance

Comparison with in-memory cached data. In order to analyze the retrieval capabilities of Umbra, we perform self-tests against a fully in-memory version of Umbra on the popular TPC-H benchmark. Although storing only the current query data is sufficient, we are restricted to scale factor 500 to fit all query data into the memory of our in-memory version. Table 2 shows the performance of the remote-only (no caching of buffer pages) and the in-memory version of our database, the end-to-end bandwidth, and the cost of the remote-only version. As mentioned, our remote-only version ignores buffered pages and retrieves all required data from remote storage. The bandwidth is computed by a sum of the retrieved data divided by the total query runtime, which serves as a lower bound.
Processing at instance bandwidth. Queries can be separated into retrieval-heavy and computation-heavy ones. The bandwidth is a good indicator for categorizing the queries. For example, Queries 1, 6, and 19 are the strongest representatives of the retrieval-heavy group. Umbra achieves an end-to-end bandwidth of up to 78 Gbit/s which is close to the limit. However, the factor between the in-memory and the remote execution time is large because Umbra could process more tuples than the network can provide.
No overhead for computationally-intensive queries. On the other hand, we observe only minor differences between the in-memory and remote-only versions for computationally intensive queries. For example, Queries 9 and 18 have a factor of $\leq 1.3\times$. Because the DBMS is at its processing limit due to intensive joins and aggregations, fetching of blocks is not very noticeable.
Effective scheduling. This shows the effectiveness of our scheduling algorithm. If the query is retrieval intensive, we saturate network bandwidth while continuing to process data. On the other hand, if Umbra is limited by computation, our scheduler does not waste CPU resources on idle downloading processes.
Spot instances. In the remote Umbra scenario, spot instances can be leveraged without any performance cliffs. However, additional safeguards need to be in place due to early instance termination. Queries affected by termination might require restarts, and commit persistence must be guaranteed.

5.2 Retrieval Manager Study

Different retrieval managers on chokepoint queries. To demonstrate the properties of our design and validate our *AnyBlob* results, we test different retrieval options within Umbra. We test our DBMS on EBS (gp3, no page cache) and on cloud object storage (no object cache). For retrieving data from S3, we implemented three different strategies. First, we use the worker threads to download their currently required object from remote storage with the AWS S3 library. The second strategy uses our asynchronous retrieval integration design, shown in Section 4, and combines it asynchronously with the AWS library. The last configuration leverages our integration and *AnyBlob* (Sections 3 and 4). To demonstrate

Table 2: In-memory and remote-only Umbra comparison demonstrates small cloud retrieval overhead (SF 500, c5n.18xlarge).

Query	GM	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22
In-Memory [s]	2.03	1.14	0.38	2.93	2.08	3.35	0.52	2.73	3.38	10.61	4.27	0.25	1.99	9.50	1.35	0.99	1.81	1.36	18.91	0.74	1.45	6.04	1.75
Remote [s]	4.94	3.52	1.97	5.87	4.18	5.77	2.47	6.41	6.86	13.34	7.68	1.14	4.74	12.47	4.15	3.97	2.42	4.63	22.20	3.82	5.06	12.24	2.54
Factor	2.42	3.08	5.16	2.01	2.01	1.72	4.78	2.35	2.03	1.26	1.80	4.58	2.39	1.31	3.07	4.01	1.34	3.41	1.17	5.15	3.50	2.03	1.45
Gbit/s	49.80	75.00	46.00	55.76	55.95	65.20	77.73	64.43	69.40	40.67	52.42	40.73	62.01	30.86	64.63	67.35	14.13	73.65	15.41	76.87	66.34	65.35	23.20
Cost S3 [¢]	0.15	0.29	0.04	0.21	0.15	0.20	0.17	0.23	0.24	0.31	0.27	0.02	0.23	0.28	0.17	0.17	0.02	0.21	0.22	0.25	0.21	0.43	0.03
Cost EC2 [¢]	0.53	0.38	0.21	0.63	0.45	0.62	0.27	0.69	0.74	1.44	0.83	0.12	0.51	1.34	0.45	0.43	0.26	0.50	2.39	0.41	0.55	1.32	0.27

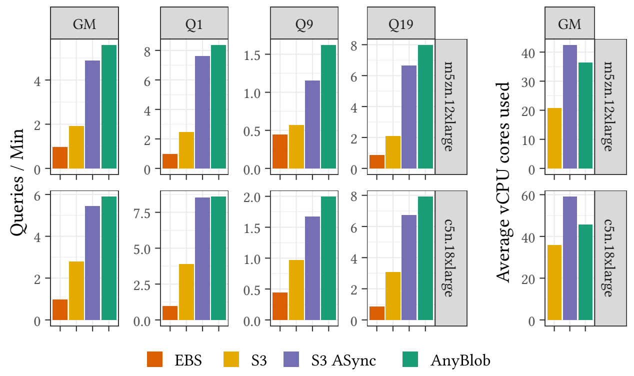


Figure 16: Internal comparison of Umbra on EBS, and on S3, + ASync (Sec. 4), + AnyBlob (Sec. 3) (SF 1000, 2 instance types).

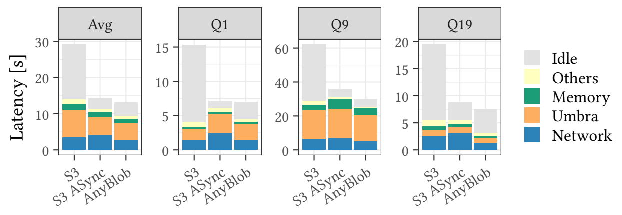


Figure 17: CPU usage traces for different networking implementations collected with Linux perf (SF 1000, c5n.18xlarge).

our cloud storage performance, all remaining experiments force Umbra to ignore columns already available in the buffer manager. Umbra always fetches these columns from remote storage.

Higher throughput while reducing CPU usage. In Figure 16, we test all TPC-H queries on two different machine types, both supporting 100 Gbit/s networking. EBS has the worst throughput due to the bandwidth limit of 1 GB/s. Asynchronous retrieval of more requests than cores is crucial for performance. By simply swapping the retrieval library from the asynchronous AWS SDK to *AnyBlob*, Umbra achieves up to a factor of 1.2× better geometric mean performance and an improvement of up to 40% on computationally expensive queries. Additionally, *AnyBlob* reduces the mean CPU usage by up to 25%. Recent trends indicate that the networking bandwidth increases faster than the number of CPU cores, making the resource usage of networking essential [5].

Retrieval requires significant CPU resources. Figure 17 breaks the query resource CPU utilization down into fine-grained tasks, such as network I/O, memory and buffer management, and processing (similar to [66]). We used perf to trace the resource utilization of different functions and aggregate the results. Umbra achieves an average CPU utilization of ~75% with asynchronous networking. Networking uses a large share of CPU time that accounts for up to 25% of the total utilization, significantly reduced by *AnyBlob*.

5.3 Scaling Properties

Thread scaling on chokepoint queries. Since our approach is highly elastic, it is very interesting to see how Umbra scales on a varying number of cores and different instances. Figure 18 shows two chokepoint queries, which we already identified in Section 5.1. The results are measured on the same instance, but we artificially reduce the amount of parallelism within our DBMS (number of worker threads). We contrast to the aforementioned in-memory version of our system. For retrieval-heavy queries (e.g., Query 1), we can see a plateau if enough cores are available to utilize the network completely. For the in-memory version, we measure a linear increase in performance until the hyper-threading boundary is reached. The performance of the computation-heavy queries (Query 9) increases as we add more cores. The remote-only Umbra version has almost the same throughput as the in-memory version.

Instance scaling. To demonstrate our scalability on different instances, we use smaller versions of the c5n.18xlarge. The c5n.9xlarge has a maximum bandwidth of 50 Gbit/s and 36 vCPUs; the c5n.4xlarge has 16 vCPUs and 25 Gbit/s bandwidth. The additional resources of larger instances improve the query runtime. Because our approach retains performance without warm caches, we can switch to larger instances as the workload increases.

5.4 End-To-End Study with Compression & AES

Workload & competitors. In this experiment, we compare the end-to-end performance on the TPC-H benchmark. To mimic a realistic OLAP scenario analyzing large amounts of data, we test scale factors (SF) of 100 (~100 GiB) and 1,000 (~1 TiB of data). Since we optimize the retrieval properties, Umbra does not cache any data to showcase our retrieval integration. We compare against Spark on a single c5n.18xlarge instance and a large warehouse of Snowflake. In 2019, Snowflake used c5d.2xlarge instances for xsmall warehouses, which was reported by a Snowflake error log [70]. Assuming this instance type for xsmall, a large warehouse would use an instance or cluster similar to our instance but with local SSDs (e.g., c5d.18xlarge or 8 × c5d.2xlarge). For Snowflake, we measure the throughput with warm cache (multiple TPC-H runs) and on another large configuration that is shut down after each query execution to enforce remote retrieval.

Fast processing from cloud storage. Figure 20 shows the performance results of different systems. As discussed in Section 4.5,

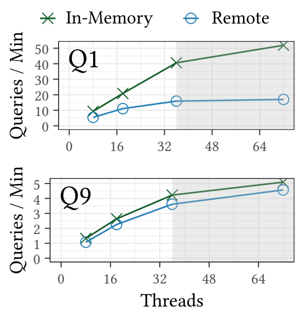


Figure 18: Scalability of queries (c5n.18xlarge).

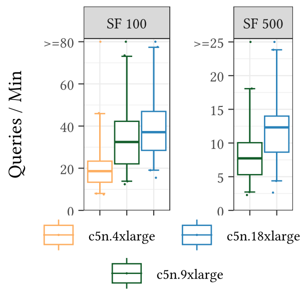


Figure 19: Scalability on different instances.

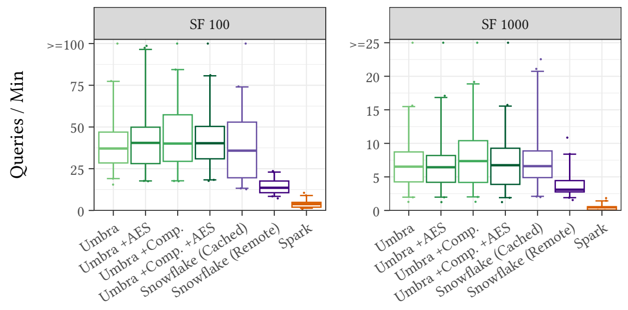


Figure 20: End-to-end system comparison on SF 100 and 1000.

Umbra is able to encrypt data automatically and implements strong compression. Compression improves performance, but encryption has a slight overhead. In a real-world scenario, we recommend using both settings for higher security without performance degradation. Although Umbra always retrieves the data from cloud storage, the performance is similar to Snowflake, which uses data caching (e.g., local SSDs). As mentioned earlier, the actual hardware configuration of Snowflake is unknown. For example, the runtime of Query 6 suggests that the instance has higher disk bandwidth than both mentioned instance settings. Clearly, these end-to-end results are influenced by the database, its execution model, and the hardware.

6 RELATED WORK

Cloud DBMS. With the dominance of the cloud for scalable solutions, many software-as-a-service database management systems emerged. Often specialized systems for either OLTP [16, 29, 30, 81] or OLAP [2, 25, 33, 58, 59, 67, 79, 87] were developed to cope with the new challenges in the cloud era [56]. Redshift [19] leverages Aqua, a computational caching layer, unaffected by resizing nodes [6]. Until recently, caching was unavoidable even for analytics dominated by the bandwidth. However, the gap between network and NVMe bandwidth is closing, making cloud storage more attractive. AWS Athena, based on Presto [73], works directly on remote data. An experimental study contrasts the architecture of these systems [77].
Processing in the cloud. Brantner et al. [27] discuss challenges and opportunities of S3 for OLTP workloads. In 2010, an experimental study provided insights into the computation power of EC2 instances; in particular, it studies the CPU resources, memory, and disk operations [72]. Our experimental study on cloud storage provides an in-depth analysis that provides all details for fast analytics on cloud storage. Leis and Kuschewski present a model for cost-optimal instance selection [54]. Although systems such as Hive and Spark can be self-hosted [78, 86], managed Hadoop is common [71].
Spot instances. Because spot instances come with huge discounts, mitigating the termination risk and hopping between instances was researched [74, 76]. Our approach is a perfect fit for spot hopping since caching is not needed for good performance. Although our experiments run faster than the termination delay of AWS (2 min) [12], a migration to another instance can retain query state [84].
Serverless computing. Serverless functions are another short-term service, which allow users to deploy resources only for the duration of a request. Since a serverless function has little memory, compute resources, and a time limit [42], many parallel function invocations are required to execute a single query. Starling [69] and Lambada [62] propose to run analytics on serverless functions. Although Lambada and Starling provide a small study on S3 in serverless environments, the characteristics are very different as these functions only have limited threads and networking (300 MiB/s), which does not require a careful retrieval design such as *AnyBlob*.
Cloud storage for DBMS. Cloud object stores attract attention as data warehouses due to their low costs. Two prominent storage solutions are Apache Iceberg and Data Lake [17, 18]. Both systems use metadata stored on the cloud object stores to provide consistent snapshots. As our storage structure is similar, our fast processing on remote data can be adapted to these storage backends. Ephemeral storage systems, such as Pocket [49], and caching for cloud storage [37, 46, 85, 89] sparked a wide variety of research. Caching solutions extend from using semantic caching on a local node [37] to leveraging spot instances as caching and offloading layer [89].
Memory disaggregation. Similar to disaggregating storage, future data centers may separate CPU from memory to improve resource flexibility. Most research finds that disaggregated memory is orthogonal to the current storage-separated design [50, 83, 90, 91].
Networking and kernel APIs. Following recent trends, future data centers will be equipped with fast Ethernet connections reaching Tbit/s [28]. OS and kernel research presents approaches to integrate these high-bandwidth network devices with low latency [28, 88]. RDMA is already explored in DBMS for fast networks with low latency [24, 47, 57, 92]. A kernel storage API study found `io_uring`, used in *AnyBlob*, to be promising [35]. Especially for fast NVMe SSDs, it is already used widespread [35, 41, 52, 55, 68].

7 CONCLUSION

This paper discusses the efficient and cost-effective usage of cloud object storage for analytics. Our first contribution is a detailed analysis on the characteristics of cloud object stores. With these insights, we developed *AnyBlob*, a modern object storage download manager based on `io_uring`. *AnyBlob* requires fewer CPU resources to achieve the same or higher throughput compared to libraries provided by cloud vendors. Finally, we demonstrated a blueprint to utilize efficient analytics on disaggregated object stores in DBMSs. Our results show that even with disabled caching, Umbra with *AnyBlob* achieves performance similar to large configurations of state-of-the-art cloud database systems that cache data locally.

REFERENCES

[1] Merv Adrian. 2022. DBMS Market Transformation 2021: The Big Picture. https://blogs.gartner.com/merv-adrian/2022/04/16/dbms-market-transformation-2021-the-big-picture/. accessed: 2022-09-30.

[2] Josep Aguilar-Saborit and Raghu Ramakrishnan. 2020. POLARIS: The Distributed SQL Engine in Azure Synapse. *Proc. VLDB Endow.* 13, 12 (2020), 3204–3216.

[3] Amazon. 2021. What's the maximum transfer speed between Amazon EC2 and Amazon S3? https://aws.amazon.com/premiumsupport/knowledge-center/s3-maximum-transfer-speed-ec2/. accessed: 2022-09-15.

[4] Amazon. 2022. Amazon S3 Storage Classes. https://aws.amazon.com/s3/storage-classes/. accessed: 2022-10-05.

[5] Amazon. 2022. Announcing Amazon EC2 C7gn instances (Preview). https://aws.amazon.com/about-aws/whats-new/2022/11/announcing-amazon-ec2-c7gn-instances-preview/. accessed: 2023-06-17.

[6] Amazon. 2022. AQUA (Advanced Query Accelerator) for Amazon Redshift. https://aws.amazon.com/redshift/features/aqua/. accessed: 2022-10-12.

[7] Amazon. 2022. AWS SDK for C++. https://github.com/aws/aws-sdk-cpp. accessed: 2022-10-05.

[8] Amazon. 2022. Encryption in transit. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/data-protection.html#encryption-transit. accessed: 2022-09-30.

[9] Amazon. 2022. Network maximum transmission unit (MTU) for your EC2 instance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/network_mtu.html. accessed: 2022-10-11.

[10] Amazon. 2022. Performance Guidelines for Amazon S3. https://docs.aws.amazon.com/AmazonS3/latest/userguide/optimizing-performance-guidelines.html. accessed: 2022-10-11.

[11] Amazon. 2022. Retrieve security credentials from instance metadata. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html#instance-metadata-security-credentials. accessed: 2022-10-15.

[12] Amazon. 2022. Spot Instance interruption notices. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html. accessed: 2022-10-08.

[13] Amazon. 2023. Amazon S3 pricing. https://aws.amazon.com/s3/pricing. accessed: 2023-06-17.

[14] Amazon. 2023. Compute optimizes instances: Network performance. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/compute-optimized-instances.html. accessed: 2023-05-02.

[15] Amazon. 2023. Filtering and retrieving data using Amazon S3 Select. https://docs.aws.amazon.com/AmazonS3/latest/userguide/selecting-content-from-objects.html. accessed: 2023-05-02.

[16] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *SIGMOD Conference.* ACM, 1743–1756.

[17] Apache. 2022. Apache Iceberg. https://iceberg.apache.org/. accessed: 2022-09-10.

[18] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *Proc. VLDB Endow.* 13, 12 (2020), 3411–3424.

[19] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD Conference.* ACM, 2205–2217.

[20] OpenSSL Project Authors. 2022. OpenSSL - Cryptography and SSL/TLS Toolkit. https://www.openssl.org/. accessed: 2022-10-15.

[21] Jens Axboe. 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. accessed: 2022-10-12.

[22] Jeff Barr. 2019. New C5n Instances with 100 Gbps Networking. https://aws.amazon.com/blogs/aws/new-c5n-instances-with-100-gbps-networking/. accessed: 2022-09-10.

[23] Jeff Barr. 2020. Amazon S3 Update Strong Read-After-Write Consistency. https://aws.amazon.com/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/. accessed: 2022-10-05.

[24] Claude Barthels, Simon Loesing, Gustavo Alonso, and Donald Kossmann. 2015. Rack-Scale In-Memory Join Processing using RDMA. In *SIGMOD Conference.* ACM, 1463–1475.

[25] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. 2022. Photon: A Fast Query Engine for Lakehouse Systems. In *SIGMOD Conference.* ACM, 2326–2339.

[26] Brendan Bouffler and Chris Liu. 2019. Deep-Dive Into 100G networking & Elastic Fabric Adapter on Amazon EC2. AWS re:Invent, https://d1.awsstatic.com/events/reinvent/2019/REPEAT_2_Deep-dive_into_100G_networking_&_Elastic_Fabric_Adapter_on_Amazon_EC2_CMP334-R2.pdf. accessed: 2022-09-10.

[27] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. 2008. Building a database on S3. In *SIGMOD Conference.* ACM, 251–264.

[28] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. 2022. Towards μs tail latency and terabit ethernet: disaggregating the host network stack. In *SIGCOMM.* ACM, 767–779.

[29] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. 2022. PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications. In *ICDE.* IEEE, 2859–2872.

[30] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD Conference.* ACM, 2477–2489.

[31] Jonathan Corbet. 2020. The rapid growth of io_uring. https://lwn.net/Articles/810414/. accessed: 2022-09-20.

[32] Craig Cotton, Henry Zhang, and Jamal Mazhar. 2019. New C5n Instances with 100 Gbps Networking. AWS re:Invent, https://www.youtube.com/watch?v=FJJxcwSfWYg. accessed: 2022-09-10.

[33] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD Conference.* ACM, 215–226.

[34] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[35] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring. In *SYSTOR.* ACM, 120–127.

[36] Dominik Durner. 2022. AnyBlob. https://github.com/durner/AnyBlob/.

[37] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: A Unified Cache Storage System for Analytical Databases. *Proc. VLDB Endow.* 14, 11 (2021), 2432–2444.

[38] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *DaMoN.* ACM, 21:1–21:3.

[39] Google. 2023. Cloud Storage pricing. https://cloud.google.com/storage/pricing. accessed: 2023-06-17.

[40] Google. 2023. Google Cloud Platform C++ Client Libraries. https://github.com/googleapis/google-cloud-cpp. accessed: 2023-06-17.

[41] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR.* www.cidrdb.org.

[42] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *CIDR.* www.cidrdb.org.

[43] Jason Hughes. 2021. Apache Iceberg: An Architectural Look Under the Covers. https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/. accessed: 2022-09-10.

[44] IBM. 2023. About IBM COS SDKs. https://cloud.ibm.com/docs/cloud-object-storage?topic=cloud-object-storage-sdk-about. accessed: 2023-06-17.

[45] IBM. 2023. Cloud Object Storage. https://cloud.ibm.com/objectstorage/create#pricing. accessed: 2023-06-17.

[46] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph (Seffi) Naor, and Sriram Rao. 2018. Netco: Cache and I/O Management for Analytics over Disaggregated Stores. In *SoCC.* ACM, 186–198.

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *OSDI.* USENIX Association, 185–201.

[48] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra. *VLDB J.* 30, 5 (2021), 883–905.

[49] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *OSDI.* USENIX Association, 427–444.

[50] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. 2022. Farview: Disaggregated Memory

with Operator Off-loading for Database Engines. In *CIDR*. www.cidrdb.org.

[51] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD Conference*. ACM, 311–326.

[52] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. In *SIGMOD Conference*. ACM.

[53] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD Conference*. ACM, 743–754.

[54] Viktor Leis and Maximilian Kuschewski. 2021. Towards Cost-Optimal Query Processing in the Cloud. *Proc. VLDB Endow.* 14, 9 (2021), 1606–1612.

[55] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *SIGMOD Conference*. ACM, 2852–2858.

[56] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.

[57] Feilong Liu, Lingyan Yin, and Spyros Blanas. 2017. Design and Evaluation of an RDMA-aware Data Shuffling Operator for Parallel Database Systems. In *EuroSys*. ACM, 48–63.

[58] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *Proc. VLDB Endow.* 3, 1 (2010), 330–339.

[59] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proc. VLDB Endow.* 13, 12 (2020), 3461–3472.

[60] Microsoft. 2023. Azure Blob Storage pricing. https://azure.microsoft.com/en-us/pricing/details/storage/blobs/. accessed: 2023-06-17.

[61] Microsoft. 2023. Azure SDK for C++. https://github.com/Azure/azure-sdk-for-cpp/. accessed: 2023-06-17.

[62] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *SIGMOD Conference*. ACM, 115–130.

[63] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.

[64] Oracle. 2023. Cloud Storage Pricing. https://www.oracle.com/cloud/storage/pricing/. accessed: 2023-06-17.

[65] Oracle. 2023. Software Development Kits. https://docs.oracle.com/en-us/iaas/Content/API/Concepts/sdks.htm. accessed: 2023-06-17.

[66] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *NSDI*. USENIX Association, 293–307.

[67] Ippokratis Pandis. 2021. The evolution of Amazon Redshift. *Proc. VLDB Endow.* 14, 12 (2021), 3162–3163.

[68] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang Won Lee. 2021. SaS: SSD as SQL Database System. *Proc. VLDB Endow.* 14, 9 (2021), 1481–1488.

[69] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *SIGMOD Conference*. ACM, 131–141.

[70] Simeon Pilgrim. 2019. What are the specifications of a Snowflake server? https://stackoverflow.com/questions/58973007/what-are-the-specifications-of-a-snowflake-server/58982398. accessed: 2023-05-02.

[71] Nicolás Poggi, Josep Lluis Berral, Thomas Fenech, David Carrera, José A. Blakeley, Umar Farooq Minhas, and Nikola Vujic. 2016. The state of SQL-on-Hadoop in the cloud. In *IEEE BigData*. IEEE Computer Society, 1432–1443.

[72] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. 2010. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. *Proc. VLDB Endow.* 3, 1 (2010), 460–471.

[73] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *ICDE*. IEEE, 1802–1813.

[74] Supreeth Shastri and David E. Irwin. 2017. HotSpot: automated server hopping in cloud spot markets. In *SoCC*. ACM, 493–505.

[75] Matt Sidley and Sally Guo. 2021. Deep dive on Amazon S3. AWS re:Invent, https://www.slideshare.net/AmazonWebServices/stg301deep-dive-on-amazon-s3-and-glacier-architecture, https://www.youtube.com/watch?v=9_vScxbIQLY. accessed: 2022-09-10.

[76] Supreeth Subramanya, Tian Guo, Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. 2015. SpotOn: a batch computing service for the spot market. In *SoCC*. ACM, 329–341.

[77] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. 2019. Choosing A Cloud DBMS: Architectures and Tradeoffs. *Proc. VLDB Endow.* 12, 12 (2019), 2170–2182.

[78] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *ICDE*. IEEE Computer Society, 996–1005.

[79] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *SIGMOD Conference*. ACM, 797–809.

[80] Daniel Vassallo. 2023. Measure Amazon S3's performance from any location. https://github.com/dvassallo/s3-benchmark. accessed: 2023-05-02.

[81] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD Conference*. ACM, 1041–1052.

[82] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *NSDI*. USENIX Association, 449–462.

[83] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. 2022. The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation. *Proc. VLDB Endow.* 16, 1 (2022), 15–22.

[84] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. On-Demand State Separation for Cloud Data Warehousing. *Proc. VLDB Endow.* 15, 11 (2022), 2966–2979.

[85] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2021. FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS. *Proc. VLDB Endow.* 14, 11 (2021), 2101–2113.

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

[87] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: Real-time OLAP Database System at Alibaba Cloud. *Proc. VLDB Endow.* 12, 12 (2019), 2059–2070.

[88] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. 2021. The Demikernel Datapath OS Architecture for Microsecond-scale Datacenter Systems. In *SOSP*. ACM, 195–211.

[89] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. CompuCache: Remote Computable Caching using Spot VMs. In *CIDR*. www.cidrdb.org.

[90] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. 2020. Rethinking Data Management Systems for Disaggregated Data Centers. In *CIDR*. www.cidrdb.org.

[91] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. 2021. Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation. *Proc. VLDB Endow.* 14, 10 (2021), 1900–1912.

[92] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD Conference*. ACM, 685–699.

PAPER P2

Crystal: A Unified Cache Storage System for Analytical Databases

Synopsis. Cloud database systems employ a tiered or disaggregated storage model, with the compute tier accessing data stored in scalable remote cloud storage, such as Amazon S3 and Azure Blobs. Many popular big data systems, including Apache *Spark* and *Greenplum*, support querying cloud storage, and cloud vendors offer specialized services to cater to this growing demand. However, the high latency and limited bandwidth to remote storage led to a renewed interest in caching technologies for analytics. These caching solutions store hot data locally in fast storage, such as SSDs. Current caching solutions work at the file or block level and employ LRU-based eviction policies. Because a single tuple of interest requires the caching of the entire block, these approaches can lead to poor storage space utilization if the data is not accessed according to the block-level partitions.

To improve cache utilization, this paper introduces a smart storage middleware, called *Crystal*. *Crystal* is decoupled from the DBMS and sits between the database system and cloud object storage, serving as a cache management system (CMS) for storage. *Crystal* runs as two components: the stand-alone *Crystal* CMS and the client connectors that are lightweight DBMS-specific adapters. The CMS manages two local caches to provide both adaptive short-term and optimized long-term caching of data. The small requested region (RR) cache is based on a traditional cache to react quickly to workload spikes, and the large oracle region (OR) cache uses an optimization algorithm for long-term knowledge.

The OR cache employs semantic caching by serving and storing single-table hyper-rectangles, called regions. A knapsack algorithm optimizes the cached regions according to the observed predicate history and considers overlap between the hyper-rectangles. Our approximative merging algorithm helps to generalize to the region of interest without overfitting. Results with Apache *Spark* and *Greenplum* show that *Crystal* has superior cache utilization, quick adaptability, and significantly enhances query performance.

Contributions. Dominik Durner contributed substantially to the content of the paper, in particular concerning the development of the proposed ideas, the implementation of the system, the evaluation, and authoring substantial parts of the paper.

Reference. Dominik Durner, Badrish Chandramouli, and Yinan Li. "Crystal: A Unified Cache Storage System for Analytical Databases". In: *PVLDB* 14.11 (2021), pp. 2432–2444

DOI. https://doi.org/10.14778/3476249.3476292

Crystal: A Unified Cache Storage System for Analytical Databases

Dominik Durner*
Technische Universität München
dominik.durner@tum.de

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Yinan Li
Microsoft Research
yinan.li@microsoft.com

ABSTRACT

Cloud analytical databases employ a disaggregated storage model, where the elastic compute layer accesses data persisted on remote cloud storage in block-oriented columnar formats. Given the high latency and low bandwidth to remote storage and the limited size of fast local storage, caching data at the compute node is important and has resulted in a renewed interest in caching for analytics. Today, each DBMS builds its own caching solution, usually based on file- or block-level LRU. In this paper, we advocate a new architecture of a smart cache storage system called *Crystal*, that is co-located with compute. Crystal's clients are DBMS-specific "data sources" with push-down predicates. Similar in spirit to a DBMS, Crystal incorporates query processing and optimization components focusing on efficient caching and serving of single-table hyper-rectangles called regions. Results show that Crystal, with a small DBMS-specific data source connector, can significantly improve query latencies on unmodified Spark and Greenplum while also saving on bandwidth from remote storage.

PVLDB Reference Format:
Dominik Durner, Badrish Chandramouli, and Yinan Li. Crystal: A Unified Cache Storage System for Analytical Databases. PVLDB, 14(11): 2432 - 2444, 2021.
doi:10.14778/3476249.3476292

1 INTRODUCTION

We are witnessing a paradigm shift of analytical database systems to the cloud, driven by its flexibility and pay-as-you-go capabilities. Such databases employ a tiered or disaggregated storage model, where the elastic *compute tier* accesses data persisted on independently scalable remote *cloud storage*, such as Amazon S3 [3] and Azure Blobs [36]. Today, nearly all big data systems including Apache Spark, Greenplum, Apache Hive, and Apache Presto support querying cloud storage directly. Cloud vendors also offer cloud services such as AWS Athena, Azure Synapse, and Google BigQuery to meet this increasingly growing demand.

Given the relatively high latency and low bandwidth to remote storage, *caching data* at the compute node has become important. As a result, we are witnessing a renewed spike in caching technology for analytics, where the hot data is kept at the compute layer in fast local storage (e.g., SSD) of limited size. Examples include the Alluxio [1] analytics accelerator, the Databricks Delta Cache [9, 15], and the Snowflake cache layer [13].

*Work done at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476292

1.1 Challenges

These caching solutions usually operate as a black-box at the file or block level for simplicity, employing standard cache replacement policies such as LRU to manage the cache. In spite of their simplicity, these solutions have not solved several architectural and performance challenges for cloud databases:

- Every DBMS today implements its own caching layer tailored to its specific requirements, resulting in a lot of work duplication across systems, reinventing choices such as what to cache, where to cache, when to cache, and how to cache.
- Databases increasingly support analytics over raw data formats such as CSV and JSON, and row-oriented binary formats such as Apache Avro [6] – all very popular in the data lake [16]. Compared to binary columnar formats such as Apache Parquet [7], data processing on these formats is slower and results in increased costs, even when data has been cached at compute nodes. At the same time, it is expensive (and often less desirable to users) to convert all data into a binary columnar format on storage, particularly because only a small and changing fraction of data is actively used and accessed by queries.
- Cache utilization (i.e., value per cached byte) is low in existing solutions, as even one needed record or value in a page makes it necessary to retrieve and cache the entire page, wasting valuable space in the cache. This is true even for optimized columnar formats, which often build per-block *zone maps* [21, 40, 48] (min and max value per column in a block) to avoid accessing irrelevant blocks. While zone maps are cheap to maintain and potentially useful, their effectiveness at block skipping is limited by the fact that even one interesting record in a block makes it necessary to retrieve it from storage and scan for completeness.
- Recently, cloud storage systems are offering predicate push-down as a native capability, for example, AWS S3 Select [4] and Azure Query Acceleration [35]. Push-down allows us to send predicates to remote storage and avoid retrieving all blocks, but exacerbates the problem of how to leverage it for effective local caching.

1.2 Opportunities

In an effort to alleviate some of these challenges, several design trends are now becoming commonplace. Database systems such as Spark are adopting the model of a plug-in "data source" that serves as an input adapter to support data in different formats. These data sources allow the *push-down* of table-level predicates to the data source. While push-down was developed with the intention of data pruning at the source, we find that it opens up a new opportunity to leverage semantics and cache data in more efficient ways.

Moreover, there is rapid convergence in the open-source community on Apache Parquet as a columnar data format, along with highly efficient techniques to apply predicates on them using LLVM with Apache Arrow [5, 8]. This opens up the possibility of system

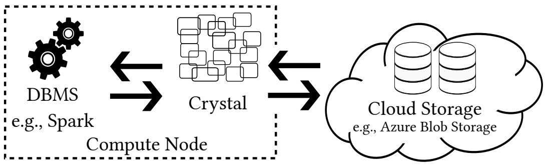


Figure 1: Crystal in big data ecosystem.

designs that perform a limited form of data processing and transformation *outside* the core DBMS easily and without sacrificing performance. Further, because most DBMSs support Parquet, it gives us an opportunity to cache data in a DBMS-agnostic way.

1.3 Introducing Crystal

We propose a new "smart" storage middleware called *Crystal*, that is decoupled from the database and sits between the DB and raw storage. Crystal may be viewed as a mini-DBMS, or *cache management system* (CMS), for storage. It runs as two sub-components:

- The Crystal CMS runs on the compute node, accessible to local "clients" and able to interact with remote storage.
- Crystal's clients, called *connectors*, are DB-specific adapters that themselves implement the data source API with push-down predicates, similar to today's CSV and Parquet data sources.

Crystal manages fast local storage (SSD) as a cache and talks to remote storage to retrieve data as needed. Unlike traditional file caches, it determines which *regions* (parts of each table) to transform and cache locally in columnar form. Data may be cached in more than one region if necessary. Crystal receives "queries" from clients, as requests with push-down predicates. It responds with local (in cache) or remote (on storage) paths for files that cover the request. The connectors pass data to the *unmodified* DBMS for post-processing as usual. Benefits of this architecture include:

- It can be shared across multiple unmodified databases, requiring only a lightweight DBMS-specific client connector component.
- It can download and transform data into automatically chosen semantic regions in the cache, in a DBMS-agnostic columnar format, reusing new tools such as Parquet and Arrow to do so.
- It can independently optimize what data to transform, filter, and cache locally, allowing multiple views of the same data, and efficiently match and serve clients at query time.

These architectural benefits come with technical challenges (Section 2 provides a system overview) that we address in this paper:

- (Sections 2 & 3) Defining an API and protocol to communicate region requests and data between Crystal and its connector clients.
- (Section 3) Efficiently downloading and transforming data to regions in the local cache, managing cache contents, and storing meta-data for matching regions with push-down predicates over diverse data types, without impacting query latency.
- (Section 4) Optimizing the contents of the cache while: (1) balancing short-term needs (e.g., a burst of new queries) vs. long-term query history; (2) handling queries that are not identical but often overlap; (3) exploiting the benefit of duplicating frequently accessed subsets of data in more than one region; and (4) taking into account the overhead incurred by creating many small files in block columnar format, instead of fewer larger ones; and (5) managing statistics necessary for the above tasks.

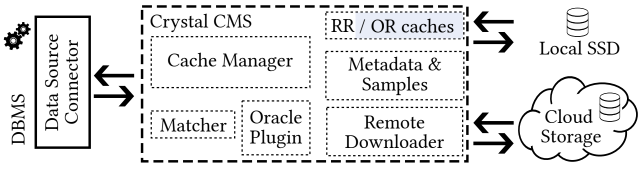


Figure 2: Crystal components.

Using Crystal, we get ***lower query latencies and more efficient use of the bandwidth between compute and storage***, as compared to state-of-the-art solutions. We validate this by implementing Crystal with Spark and Greenplum connectors (Section 5). Our evaluation using common workload patterns shows Crystal's ability to outperform block-based caching schemes with lower cache sizes, improve query latencies by up to 20x for individual queries (and up to 8x on average), adapt to workload changes, and save bandwidth from remote storage by up to 41% on average (Section 6).

We note that Crystal's cached regions may be considered as materialized views [20, 27, 44, 45] or semantic caches [14, 29, 30, 41–43, 47], thereby inheriting from this rich line of work. Our caches have the additional restriction that they are strictly the result of single-table predicates (due to the nature of the data source API). Specifically, Crystal's regions are disjunctions of conjunctions of predicates over each individual table. This restriction is exploited in our solutions to the technical challenges, allowing us to match better, generalize better, and search more efficiently for the best set of cached regions. As data sources mature, we expect them to push down cross-table predicates and aggregates in future, e.g., via *data-induced predicates* [28]. Such developments will require a revisit of our algorithms in future; for instance, our region definitions will need to represent cross-table predicates. We focus on read-only workloads in this paper; updates can be handled by view invalidation (easy) or refresh (more challenging), and are left as future work. Finally, we note that Crystal can naturally benefit from remote storage supporting push-down predicates; a detailed study is deferred until the technology matures to support columnar formats natively (only CSV files are supported in the current generation). We cover related work in Section 7 and conclude in Section 8.

2 SYSTEM OVERVIEW

Figure 1 shows where Crystal fits in today's cloud analytics ecosystem. Each compute node runs a DBMS instance; Crystal is co-located on the compute node and serves these DBMS instances via data source connectors. The aim is to serve as a caching layer between big data systems and cloud storage, exploiting fast local storage in compute nodes to reduce data accesses to remote storage.

2.1 Architecture

A key design goal is to make Crystal sufficiently generic so that it can be plugged into an existing big data system with minimum engineering effort. Therefore, Crystal is architected as two separate components: a light DBMS-specific data source connector and the Crystal CMS process. These are described next.

2.1.1 Data Source Connector. Modern big data systems (e.g., Spark, Hive, and Presto) provide a *data source API* to support a variety of data sources and formats. A data source receives push-down filtering and column pruning requests from the DBMS through this API.

Thus, the data source has the flexibility to leverage this additional information to reduce the amount of data that needs to be sent back to the DBMS, e.g., via block-level pruning in Parquet. In this paper, we refer to such push-down information as a *query* or *requested region*. A Crystal connector is integrated into the unmodified DBMS through this data source API. It is treated as another data source from the perspective of the DBMS, and as a client issuing queries from the perspective of the Crystal CMS.

2.1.2 Crystal CMS. Figure 2 shows the Crystal CMS in detail. It maintains two local caches – a small *requested region* (RR) cache and a large *oracle region* (OR) cache – corresponding to short- and long-term knowledge respectively. Both caches store data in an efficient columnar open format such as Parquet. Crystal receives "queries" from connectors via the Crystal API. A query consists of a request for a file (remote path) with push-down predicates. Crystal first checks with the `Matcher` to see if it can cover the query using one or more cached regions. If yes (cache hit), it returns a set of file paths from local storage. If not (cache miss), there are two options:

(1) It responds with the remote path so that the connector can process it as usual. Crystal optionally requests the connector to store the downloaded and filtered region in its RR cache.
(2) It downloads the data from remote, applies predicates, stores the result in the RR cache, and returns this path to the connector.

Thus, the RR cache is populated *eagerly* by either Crystal or the DBMS. Not every requested region is cached eagerly; instead an LRU-2 based decision is taken per request.

More importantly, in the background, Crystal collects a historical trace of queries and invokes a caching `Oracle Plugin` module to compute the best content for the OR cache. The new content is populated using a combination of remote storage and existing content in the RR and OR caches. Section 3 covers region processing in detail, while Section 4 covers cache optimization.

2.2 Generality of the Crystal Design

As mentioned above, Crystal is architected with a view to making it easy to use with any cloud analytics system. Crystal offers three extensibility points. First, users can replace the caching oracle with a custom implementation that is tailored to their workload. Second, the remote storage adapter may be replaced to work with any cloud remote storage. Third, a custom connector may be implemented for each DBMS that needs to use Crystal.

The connector interfaces with Crystal with a generic protocol based simply on file paths. Cached regions are stored in an open format (Parquet) rather than the internal format of a specific DBMS, making it DBMS-agnostic. Further, a connector can feed the cached region to the DBMS by simply invoking its built-in data source for the open format (e.g., the built-in Parquet reader in Spark) to read the region. Thus, the connector developer does not need to manually implement the conversion, making its implementation a fairly straightforward process. In Section 5, we discuss our connectors for Spark and Greenplum, which take less than 350 lines of code.

2.3 Revisiting the Caching Problem

Leveraging push-down predicates, Crystal caches different subsets of data called regions. Regions can be considered as views on the table, and are a form of semantic caching [14, 29, 30, 42, 43, 47]. Compared to traditional file caching, the advantage of semantic caching is two-fold. First, it usually returns a much tighter view to the DBMS, and thus reduces the need to post-process the data, saving I/O and CPU cost. Second, regions can be much smaller than the original files, resulting in better cache space utilization and higher hit ratios. For example, Figure 3 shows a case where regions capture all views of all queries, whereas LRU-based file caching can only keep less than half of these views.

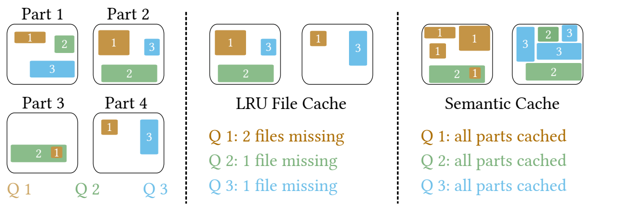


Figure 3: Benefit of semantic vs. traditional file caching. The DBMS schedules Q1, Q2, and Q3 more frequently. Only the semantic cache can answer these without remote access.

Cached regions in Crystal may overlap. In data warehouses and data lakes, it is common to see that a large number of queries access a few tables or files, making overlapping queries the norm rather than the exception at the storage layer. Therefore, Crystal has to take overlap into account when deciding which cached data should be evicted. To the best of our knowledge, previous work on replacement policies for semantic caching does not consider overlap of cached regions (see more details in Section 7).

With overlapping views, the replacement policy in Crystal becomes a very challenging optimization problem (details in Section 4). Intuitively, when deciding if a view should be evicted from the cache, all other views that are overlapping with this view should also be taken into consideration. As a result, traditional replacement policies such as LRU that evaluate each view independently are not suitable for Crystal, as we will show in the evaluation (Section 6).

Recall that we split the cache into two regions: requested region (RR) and oracle region (RR). The OR cache models and solves the above problem as an optimization problem, which aims to find the nearly optimal set of overlapping regions that should be retained in the cache. Admittedly, solving the optimization problem is expensive and thus cannot be performed on a per-request basis. Instead, the OR cache recomputes its contents periodically, and thus mainly targets queries that have sufficient statistics in history. In contrast, the RR cache is optimized for new queries, and can react immediately to workload changes. Intuitively, the RR cache serves as a "buffering" region to temporarily store the cached views for recent queries, before the OR cache collects sufficient statistics to make longer-term decisions. This approach is analogous to the C-Store architecture [46], where a writable row store is used to absorb newly updated data before it is moved to a highly optimized column store in batches. Collectively, the two regions offer an efficient and reactive solution for caching.

3 REGION PROCESSING

In this section, we focus on region matching and the creation of cached regions. Before we explain the details of the process of

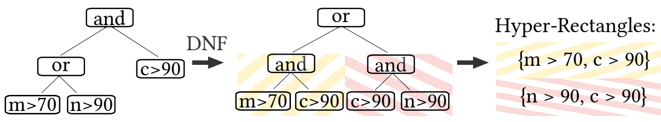


Figure 4: Transforming any predicate tree into unions of hyper-rectangles.

creating regions and matching cached regions to requests, we first show how to transform client requests into region requests.

3.1 API

Crystal acts as a storage layer of the DBMS. It runs outside the DBMS and transfers information via a minimalistic socket connection and shared space in the filesystem (e.g., SSDs, ramdisk). During a file request, the DBMS exchanges information about the file and the required region with Crystal. Because access to remote files is expensive, Crystal tries to satisfy the request with cached files.

The overall idea is that Crystal overwrites the accessed file path such that the DBMS is pointed to a local file. For redirecting queries, Crystal relies on query metadata such as the file path, push-down predicates, and accessed fields. Crystal evaluates the request and returns a cached local file or downloads the requested file. Afterward, the location of the local file is sent to the DBMS which redirects the scan to this local file. Crystal guarantees the completeness for a given set of predicates and fields. Internally, Crystal matches the query metadata with local cache metadata and returns a local file if it satisfies the requirements.

We use a tree string representation for push-down predicates in our API. Since predicates are conventionally stored as an AST in DBMS, we traverse the AST to build the string representation. Each individual item uses the syntax similar to *operation(left, right)*. We support binary operators, unary operators, and literals which are the leaf nodes of the tree. The binary operation is either a combination function of multiple predicates (such as *and*, *or*) or an atomic predicate (such as *gt*, *lt*, *eq*, ...). Atomic predicates use the same binary syntax form in which *left* represents the column identifier and *right* the compare value. To include the negation of sub-trees, our syntax allows *operation(exp)* with the operation *not*.

3.2 Transformation & Caching Granularity

Crystal receives the string of push-down predicates and transforms it back to an internal AST. Because arguing on arbitrarily nested logical expressions (with *and* and *or*) is hard, Crystal transforms the AST to Disjunctive Normal Form (DNF). In the DNF, all conjunctions are pushed down into the expression tree, and conjunctions and disjunctions are no longer interleaved. In Crystal, regions are identified by their disjunction of conjunctions of predicates. Regions also contain their sources (i.e., the remote files) and the projection of the schema. This allows us to easily evaluate equality, superset, and intersection between regions which we show in Section 3.3.

The construction of the DNF follows two steps. First, all negations are pushed as far as possible into the tree which results in Negation Normal Form (NNF). Besides using the De-Morgan rules to push down negations, Crystal pushes the negations inside the predicates. For example, *not(lt(id, 1))* will be changed to *gteq(id, 1)*.

After receiving the NNF, Crystal distributes conjunctions over disjunctions. The distributive law pushes *or*s higher up in the tree which results in the DNF. It transforms *and(a, or(b, c))* to *or(and(a, b), and(a, c))*. Although this algorithm could create 2^n leaves in theory, none of our experiments indicate issues with blow-up.

Because the tree is in DNF, the regions store the pushed-down conjunctions as a list of column restrictions. These conjunctions of restrictions can be seen as individual geometric hyper-rectangles. Regions are fully described by the disjunction of these hyper-rectangles. Figure 4 shows the process of creating the DNF and extracting the individual hyper-rectangles. Although we use the term hyper-rectangles, the restrictions can have different shapes. Crystal supports restrictions, such as *noteq*, *isNull*, and *isNotNull*, that are conceptually different from hyper-rectangles.

Crystal's base granularity of items is on the level of regions, thus all requests are represented by a disjunction of conjunctions. However, individual conjunctions of different regions can be combined to satisfy an incoming region request. Some previous work on semantic caching (e.g., [14, 17]) considers only non-overlapping hyper-rectangles. Non-overlapping regions can help reduce the complexity of the decision-making process. Although this is desirable, non-overlapping regions impose additional constraints.

Splitting the requests into sets of non-overlapping regions is expensive. In particular, the number of non-overlapping hyper-rectangles grows combinatorial. To demonstrate this issue, we evaluated three random queries in the lineitem space which we artificially restrict to 8 dimensions [23]. If we use these three random hyper-rectangles as input, 16 hyper-rectangles are needed to store all data non-overlapping. This issue arises from the number of dimensions that allow for multiple intersections of hyper-rectangles. Each intersection requires the split of the rectangle. In the worst case, this grows combinatorial in the number of hyper-rectangles.

Because all extracted regions need statistics during the cache optimization phase, the sampling of this increased number of regions is not practical. Further, the runtime of the caching policies is increased due to the larger input which leads to outdated caches.

Moreover, smaller regions require that more cached files are returned to the client. Figure 5 shows that each additional region incurs a linear overhead of roughly 50ms in Spark. The preliminary experiment demonstrates that splitting is infeasible due to the combinatorial growth of non-overlapping regions. Therefore, Crystal does not impose restrictions on the semantic regions themselves. This raises an additional challenge during the optimization phase of the oracle region cache, which we address in Section 4.5.

3.3 Region Matching

With the disjunction of conjunctions, Crystal determines the relation between different regions. Crystal detects equality, superset, intersections, and partial supersets relations. Partial supersets contain a non-empty number of conjunctions fully.

Crystal uses intersections and supersets of conjunctions to argue about regions. Conjunctions contain restrictions that specify the limits of a column. Every conjunction has exactly one restriction for each predicated column. Restrictions are described by their column identifier, their range (*min, max*), their potential equal value, their set of non-equal values and whether *isNull* or *isNotNull* is set. If two

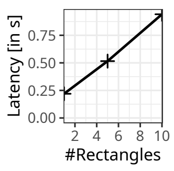


Figure 5: Many regions overhead.

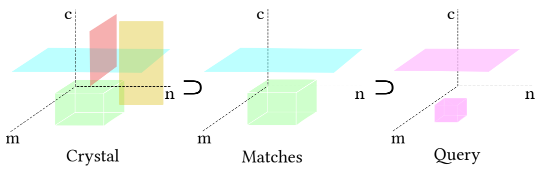


Figure 6: Crystal matches Query.

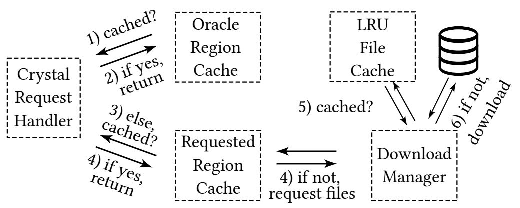


Figure 7: Request handling.

restrictions p_x and p_y are on the same column, Crystal computes if p_x completely satisfies p_y or if p_x has an intersection with p_y. For determining the superset, we first check if the null restrictions are not contradicting. Second, we test whether the (*min, max*) interval of p_x is a superset of p_y. Afterward, we check whether p_x has restricting non-equal values that discard the superset property and if all additional equal values of p_y are also included in p_x.

For two conjunctions c_x and c_y, $c_x \supset c_y$ if c_x only contains restrictions that are all less restrictive than the restrictions on the same column of c_y. Thus, c_x must have an equal number or fewer restrictions which are all satisfying the matched restrictions of c_y. Otherwise, $c_x \not\supset c_y$. c_x can have fewer restrictions because the absence of a restriction shows that the column is not predicated.

In the following, we show the algorithms to determine the relation between two regions r_x and r_y.

- $r_x \supset r_y$ holds if all conjunctions of r_y find a superset in r_x.
- $r_x \cap r_y \neq \emptyset$ holds if at least one conjunction of r_x finds an intersecting conjunction of r_y.
- $\exists \text{ conj} \subset r_x : \text{ conj} \subset r_y$ (partial superset) holds if at least one conjunctions of r_y finds a superset in r_x.
- $r_x = r_y$: $r_x \supset r_y \wedge r_y \supset r_x$

Figure 6 shows an example that matches a query that consists of two hyper-rectangles to two of the stored regions.

3.4 Request Matching

During region requests, Crystal searches the caches to retrieve a local superset. Figure 7 shows the process of matching the request. First, the oracle region cache is scanned for matches. If the request is not fully cached, Crystal tries to match it with the requested region cache. If the query was not matched, the download manager fetches the remote files (optionally from a file cache).

During the matching, a full superset is prioritized. Only if no full superset is found, Crystal tries to satisfy the individual conjunctions. The potential overlap of multiple regions and the overhead shown in Section 3.2 are the reasons to prefer full supersets. If an overlap is detected between A and B, Crystal needs to create a reduced temporary file. Otherwise, tuples are contained more than once which would lead to incorrect results. For example, it could return A and $B - A$ to the client. The greedy algorithm, presented in Algorithm 1 reduces the number of regions if multiple choices are possible. We choose the region that satisfies most of the currently unsatisfied conjunctions and continue until all have been satisfied.

We optimize the matching of regions by partitioning the cache according to the remote file names and the projected schema. The file names are represented as (bit-)set of the remote file catalog. This set is sharded by the tables. Similarly, the schema can be represented as a (bit-)set. The partitioning is done in multiple stages. After the

Algorithm 1: Greedy reduction of multiple matches

```
input   :Region requestedRegion, List<Regions> partialMatches
output  :List<Regions> regions
BitSet<requestedRegion.disjunctionCount> matches(0);
while true do
    if matches.isAllBitsSet() then
        return regions
    bestRegion = {}; bestVal = 0
    foreach p ∈ partialMatches do
        curval = additionalMatches(p, matches)
        if curVal > bestVal then
            bestRegion = p; bestVal = curVal
    if !bestRegion then return {}
    partialMatches = partialMatches \ bestRegion
    regions = regions ∪ buildTempFile(bestRegion, regions)
    matches.setAll(requestedRegion.satisfiedConjunctions(bestRegion))
```

fast file name superset check, all resulting candidates are tested for a superset of the schema. Only within this partition of superset regions, we scan for a potential match. Although no performance issues arise during region matching, multi-dimensional indexes (e.g., R-trees) can be used to further accelerate lookups.

3.5 Creating Regions

The cached regions of Crystal are stored as Apache Parquet files. Crystal leverages Apache Arrow for reading and writing snappy encoded Parquet files. Internally, Parquet is transformed into Arrow tables before Crystal creates the semantic regions.

Gandiva, which is a newly developed execution engine for Arrow, uses LLVM compiled code to filter Arrow tables [8]. As this promises superior performance in comparison to executing tuple-at-a-time filters, Crystal translates its restrictions to Gandiva filters. When Crystal builds new Parquet files to cache, the filters are compiled to LLVM and executed on the in-memory Arrow data. Afterward, the file is written to disk as snappy compressed Parquet file. If a file is accessed the first time, Crystal creates a sample that is used to predict region sizes and to speed up the client's query planning.

3.6 Client Database Connector

Database systems are often able to access data from different formats and storage layers. Many systems implement a connection layer that is used as an interface between the DBMS and the different formats. For example, Spark uses such an abstraction layer - known as data source.

Crystal is connected to the DBMS by implementing such a small data source connector. As DBMSs can process Parquet files already, we can easily adapt this connector for Crystal. Crystal interacts with the DBMS via a socket connection and transfers files via shared disk space or ramdisk. Since Crystal returns Parquet files, the DBMS can already process them without any code modifications.

The only additional implementation needed is the exchange of control messages. These consist of only three different messages and the responses of Crystal. One of the messages is optional and is used to speed up query planning. The scan request message and the message that indicates that a scan has finished are required by all Crystal clients. The first message includes the path of the remote file, the push-down predicates, and the required fields of the schema. Crystal replies with a collection of files that can be used instead of the original remote file. The finish message is required to delete cached files safely that are no longer accessed by the client. The optional message inquires a sample of the original data to prevent storage accesses during query planning.

3.7 Cloud Connection

Crystal itself also has an interface similar to the data source. This interface is used to communicate with various cloud connectors. The interface implements simple file operations, such as listings of directories and accesses to files. For blob storage, the later operation basically downloads the file from remote storage to the local node.

Recently, cloud providers have been adding predicate push-down capabilities to their storage APIs, e.g., S3 Select [4]. Clients can push down filters to storage and receive the predicated subset. This feature can incur additional monetary costs, as well as a per-request latency. Crystal complements this feature naturally, as it is aware of semantic regions and can use predicate push-down to populate its cache efficiently. As Crystal can reuse cached results locally, it can save on future push-down costs as well.

Crystal implements a download manager that fetches blobs from remote and stores them into ramdisk. The client is pointed to this location, and as soon as it finishes accessing it, the file is deleted again. Multiple accesses can be shared by reference counting.

4 CACHE OPTIMIZATION

This section summarizes the architecture of our caches, followed by more details on caching. Finally, we explain our algorithms that explore and augment the overlapping search space.

4.1 Requested Region and Oracle Region Cache

Recall that Crystal relies on two region caches to capture short- and long-term trends. The *RR* cache is an eager cache that stores the result of recently processed regions. The long-term insights of the query workload are captured by the *OR* cache. This cache leverages the history of region requests to compute the ideal set of regions to cache locally for best performance. Crystal allows users to plug-in a custom oracle; we provide a default oracle based on a variant of Knapsack (covered later). After the oracle determines a new set of regions to cache, Crystal computes these regions in the background and updates the *OR* cache. The creation in the background allows to schedule more expensive algorithms (runtime) to gather meaningful insights. This allows for computing (near-) optimal results and the usage of machine learning in future work. The oracle runs in low priority, consuming as little CPU as possible during high load.

An interesting opportunity emerges from the collaboration between the two caches. If the *OR* cache decided on a set of long-term relevant regions, the requested region cache does not need to compute any subset of the already cached long-term regions. On the other hand, if the requested region cache has regions that are considered for long-term usage, the *OR* cache can take control over these regions and simply move them to the new cache.

4.2 Metadata Management

A key component for predicting cached regions is the history of requested regions. To recognize patterns, the previously accessed regions are stored within Crystal. We use a ring-buffer to keep the most recent history. Each buffer element represents a single historic region request which has been computed by a collection of (remote) data files. These files are associated with schema information, tuple count, and size. The selectivity of the region is captured by result statistics. The database can either provide result statistics, or Crystal will compute them. Crystal leverages previously created samples to generate result statistics. In conjunction with the associated schema information, Crystal predicts the tuple count and the result size.

4.3 Oracle Region Cache

Long-term trends are detected by using the oracle region cache. An oracle decides according to the seen history which regions need to be created. The history is further used as a source of candidate regions that are considered to be cached.

The quality of the cached items is evaluated with the recent history of regions. Each cached region is associated with a benefit value. This value is the summation of bytes that do not need to be downloaded if the region is stored on the DBMS node. In other words, how much network traffic is saved by processing the history elements locally. Further, we need to consider the costs of storing candidate regions. The costs of a region are simply given by the size it requires to be materialized. The above caching problem can be expressed as the knapsack problem: maximize $\sum_{i=1}^{n} b_i x_i$ subject to $\sum_{i=1}^{n} w_i x_i \leq W$ where $x_i \in \{0, 1\}$. The saved bandwidth by caching a region is denoted by b, the size of the materialized cache by w. If the region is picked $x = 1$, otherwise $x = 0$. The goal is to maximize the benefit while staying within the capacity W.

However, the current definition cannot capture potential overlap in regions well. As the benefit value is static, history elements that occur in multiple regions would be added more than once to the overall value. Thus the maximization would result in a suboptimal selection of regions. In Section 4.5, we show the adaptations of our proposed algorithm to compensate for the overlapping issue.

4.4 Knapsack Algorithms

Dynamic programming (DP) can be used to solve the knapsack optimally in pseudo-polynomial time. The most widespread algorithm iterates over the maximum number of considered items and the cache size to solve the knapsack optimal for each sub-problem instance. Combining the optimally solved sub-problems results in the optimal knapsack, but the algorithm lies in the complexity of $O(n * W)$. Another possible algorithm iterates over the items and benefit values, and lies in $O(n * B)$ (B denotes maximum benefit).

In our caching scenario, we face two challenges with the DP approach. First, both W (bytes needed for storing the regions) and B (bytes the cached element saves from being downloaded) are large. Relaxing these values by rounding to mega-bytes or giga-bytes reduces the complexity, however, the instances are not solved

Algorithm 2: Overlap Greedy Knapsack

```
input  : List<Region> history, List<Region> candidates, Int maxCacheSize
output : List<Region> cache
List<Region> cache = List<Region>(); Int currentCacheSize = 0
Map<Float, Region> benefitRatioMap = evaluate(candidates, history, cache)
foreach {benefit, region} $\in$ benefitRatioMap do
    if currentCacheSize + region.size > maxCacheSize then
        return cache
    foreach item $\in$ cache do
        if item $\subseteq$ region then
            cache = cache \ item
    cache = cache $\cup$ region
    benefitRatioMap = evaluate(candidates, history, cache)
    currentCacheSize += region.size
return cache
```

optimally anymore. Second, the algorithm considers that each sub-problem was solved optimally. To solve the overlapping issue, only one region is allowed to take the benefit of a single history element. An open question is to decide which sub-problem receives the benefit of an item that can be processed with several regions.

Since many knapsack instances face a large capacity W and unbound benefit B, approximation algorithms were explored. In particular, the algorithm that orders items according to the benefit-cost ratio has guaranteed bounds and a low runtime complexity of $O(n * log(n))$. The algorithm first calculates all benefit ratios $v = \frac{b}{w}$ and orders the items accordingly. In the next step, it greedily selects the items as long as there is space in the knapsack. Thus, the items with the highest cost to benefit ratio v are contained in the knapsack. This algorithm solves the relaxed problem of the fractional knapsack optimal which loosens $x \in \{0, 1\}$ to $x \in [0, 1]$ [24].

4.5 Overlap-aware Greedy Algorithm

This greedy knapsack algorithm is used as the basis of our adaptations. In contrast to DP, this approach gives us an order of the picked items which allows us to incorporate the benefit changes.

Algorithm 2 shows the adapted greedy knapsack algorithm. The general idea is that we recompute the benefit ratio for each picked item. For each iteration step, we reevaluate the benefit and size of the current candidate set. The evaluation function sorts the input according to this benefit ratio. Thus, regions that result in higher returns in comparison to the caching size are picked earlier. Note that we only consider regions that have a benefit ratio > 1 to reduce unnecessary computation for one-time requests. The runtime complexity of the adapted algorithm is $O(n^2 * log(n))$.

The evaluation of the benefit ratio is adapted according to the previously chosen regions. We define three geometric rules which change the ratio of unpicked elements.

(1) if a candidate is a superset of a picked item, we reduce the weight and the benefit by the values of the picked elements.
(2) if a candidate is a subset of an already picked item, we reduce the benefit to 0 as it does not provide any additional value.
(3) if a candidate is intersected with an already picked item, we reduce the benefit by the history elements that are covered completely by both regions.

(1) A container region $r_c = \{r_1, r_2, \ldots, r_n, r_x\}$ fully contains n stand-alone regions and the remainder region r_x. The cost of r_c is computed by $w_c = w_x + \sum_{i=1}^{n} w_i$ and the benefit $b_c = b_x + \sum_{i=1}^{n} b_i$. If a region r_k is fully contained in another region r_c, we reduce both

Algorithm 3: Approximative Merging Augmentation

```
input  : List<Region> history, Int maxRegions, Int maxSize, Int maxCacheSize
output : List<Region> resultRegions
// RegionStruct consists of Region, Quality (0), and Size Savings (0)
  List< RegionStruct<Region, Int, Int> > enlargedRegions
foreach $r \in$ history do
    foreach $r' \in$ history \ {$r_0, \ldots, r$} do
        $r$.enlargeAll($r'$, enlargedRegions)
foreach $r \in$ enlargedRegions do
    foreach $r' \in$ history do
        if $r$.region.satisfies($r'$) then
            $r$.quality += 1; $r$.sizeSavings += $r'$.size
sort(enlargedRegions, $\lambda$ (r1, r2) { r1.quality > r2.quality })
while !enlargedRegions.empty() $\land$ maxRegions > 0 do
    $r$ = enlargedRegions.pop(); considered = true
    foreach $r' \in$ resultRegions do
        if r'.satisfies($r$.region) $\land$ $r'$.size < maxSize then
            considered = false
    if !considered then
        continue
    $r$.region.computeStatisticsWithSample()
    if $r$.region.size < maxSize $\lor$ ($r$.region.size < $r$.sizeSavings $\land$ $r$.region.size <
      maxCacheSize) then
        resultRegions = resultRegions $\cup$ $r$.region; maxRegions -= 1
return resultRegions
```

the weight and benefit of r_c when r_k is picked. Thereby, we simulate r_c' which is a non overlapping version of r_c with $v_k >= v_c >= v_{c'}$. In the case, the greedy algorithm picks r_c' in a future iteration, we actually add r_c and remove the previously picked item r_k.

(2) If r_c is picked, all the other included regions in r_c are fully contained with their benefits and weights. Since the greedy algorithm picks $r_c \Rightarrow \forall r \in r_c : v_c >= v_r$. The benefit of all contained r is reduced to 0 as all history elements are included in r_c.

(3) Besides full containment, regions can have partial overlap. Assume that r_x and r_y overlap partially, and r_x is picked. Our algorithm reduces the benefit b_y by all history elements that are covered by both r_x and r_y. However, we cannot reduce the costs of caching r_y as we would need to compute the non-overlapping part of the regions. This is in direct contradiction to the goal of minimizing region splits as shown in Section 3.2. For retaining optimality, all interleaving regions must be considered as the potentially picked item in an individual branch of the problem. The branch that yields the maximum benefit is chosen as the winner. Unfortunately, this introduces exponential growth of the search space. Our experiments show that even without considering all paths, our greedy algorithm produces highly effective *OR* caches. Although this revokes the fractional knapsack optimality guarantee, our greedy algorithm only picks the locally optimal choice and does not branch.

4.6 Region Augmentation

To predict regions that are accessed in the future, the oracle needs to generalize. If the candidate set of the decision-making solely consists of the seen history elements, the oracle will overfit. Thus, a crucial part is the augmentation of the candidate set to include unseen regions that are evaluated according to the seen history.

To find generalized candidate sets, we developed the approximative merging algorithm. This algorithm tries to merge intersecting regions to find the generalized region of interest. In particular, we combine two predicates and for each predicate the global min and global max are used as new dimension restrictions. As this introduces n^2 new regions, we only merge conjunctions if they intersect in at least one dimension. To overcome the issue of non-intersecting but neighboring hyper-rectangles (e.g., $x < 1, x \geq 1$), we allow for approximative intersections that add a small delta to the boundaries.

The full approximative merging procedure is presented in Algorithm 3. First, we compute enlarged regions from the history and consider the ones that match the previously described criteria. After determining new enlarged regions, each enlarged region is assigned a quality and size saving value. Quality counts how many history regions can be processed with this enlarged region. The overall sum of the size required by each region, that can be processed with this new enlarged region, denotes the size saving. With these properties, Crystal ranks the new regions according to quality and adds the highest ranked ones to the candidate set. We only add new regions if these cannot be represented by already existing regions and their size overhead is either smaller than a defined maximum size or the size saving is larger than the region itself. The sizes of the enlarged regions are computed with the help of the samples already collected for each file. In the experimental evaluation, we add at most 20% of additional regions (according to the history size) and define a maximum size of 20% of the total semantic cache size.

4.7 Requested Region Cache

The requested region cache is similar to a traditional cache but with semantic regions instead of pages. It decides in an online fashion whether the requested region should be cached. The algorithm must be simple to reduce decision latencies. Traditional algorithms, such as LRU and its variants, are good fits in terms of accuracy and efficiency. Besides the classic LRU cache, experiments showed the benefit of caching regions after the second (k-th in general) occurrence. With the history already available for *OR*, this adaption is simple and does not introduce additional latency. For combined *OR* and *RR* with *LRU-k*, it is beneficial to reduce the history size by the *RR/OR* split as long-term effects are captured by *OR*.

One of the biggest advantages of the *RR* cache is the fast reaction to changes in the queried workload. In comparison to the *OR* cache that only refreshes periodically, the request cache is updated constantly. This eager caching, however, might result in overhead due to additional writing of the region file. To overcome this issue, the client DBMS can simultaneously work on the raw data and provide the region as a file for Crystal; this extension is left as future work.

5 IMPLEMENTATION DETAILS

Crystal is implemented as a stand-alone and highly parallel process that sits between the DBMS and blob storage. This design helps to accelerate workloads across different database systems. Crystal is a fully functional system that works with diverse data types and query predicates, and is implemented in C++ for optimal performance.

Parallel Processing within Crystal. Latency critical parts of Crystal are optimized for multiple connections. Each new connection uses a dedicated thread for building the predicate tree and matching cached files. If a file needs to be downloaded, it is retrieved by a pool of download threads to saturate the bandwidth. All operations are either implemented lock-free, optimistically, or with fine-grained shared locks. Liveness of objects and their managed cached files is tracked with smart pointers. Therefore, Crystal parallelizes well and can be used as a low latency DBMS middleware.

Crystal also handles large files since some systems do not split Parquet files into smaller chunks. During matching we recognize which parts of the original file would have been read and translate it to the corresponding region in the cached files. Further, we are able to parallelize reading and processing Parquet files.

Spark Data Source. For our evaluation, we built a data source to communicate between Spark and Crystal, by extending the existing Parquet connector of Spark with less than 350 lines of Scala code. The connector overrides the scan method of Parquet to retrieve the files suggested by Crystal. Because Spark pushes down predicates to the data source, we have all information available for using the Crystal API. As Spark usually processes one row iterator per file, we developed a meta-iterator that combines multiple file iterators transparently (Crystal may return multiple regions). The connector is packaged as a small and dynamically loaded Java jar.

Greenplum Data Source. Further, we built a connector for Greenplum which is a cloud scale PostgreSQL derivative with an external extension framework – called PXF [34, 51]. PXF allows one to access Parquet data from blob storage [52]. We modified the Parquet reader such that it automatically uses Crystal if available. Our changes to the Greenplum connector consist of less than 150 lines of code. Without recompiling the core database, Crystal accelerates Greenplum by dynamically attaching the modified PXF module.

Both connectors currently do not support sending regions back to Crystal; instead, Crystal itself handles additions to the RR cache.

Azure Cloud Connection. We use Azure Blob Storage to store remote data, using a library called `azure-storage-cpplite` [37] to implement the storage connector. The library just translates the file accesses to CURL (HTTPS) requests. Other cloud providers have similar libraries with which connections can be easily established. Crystal infers the cloud provider from the remote file path. The file path also gives insights into the file owner (user with pre-configured access token) and the blob container that includes the file.

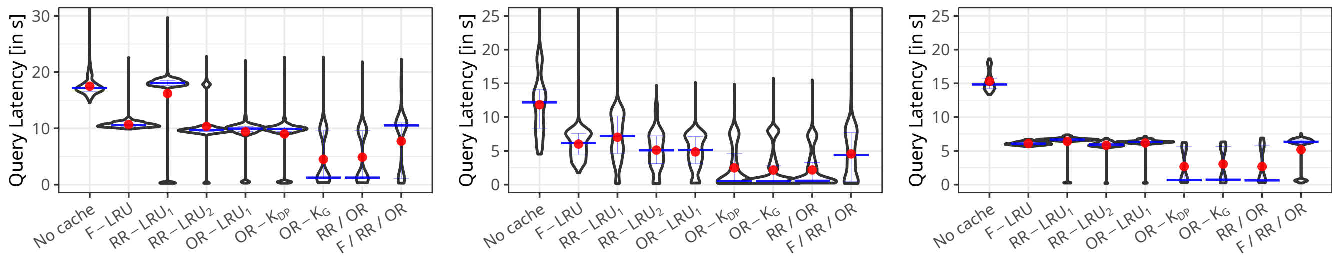
6 EXPERIMENTAL EVALUATION

In this section, we evaluate Crystal as an acceleration unit for Spark, and further report an experiment with Greenplum to show Crystal's generality. Our experiments utilize a single compute node that is connected to standard Azure Blob Storage. The blob storage uses the pre-selected configuration of standard storage and hot tier. All experiments were performed on the DS14_v2 virtual machine. This instance features 16 cores with 112 GB main memory. It comes with 224 GB premium (SSD) storage attached and has a maximum network bandwidth of 12 Gbit/s. The Apache Spark experiments run on version 3.0.1 pre-built for Hadoop 3.2. Our software stack includes Apache Arrow 3.0.0 and azure-storage-cpplite (be490ed).

6.1 Datasets and Caching Strategies

We test workloads comprising real-world data and benchmarks. Following prior work [18], we synthesize queries that contain a mix of range filters and equality filters. Each query is associated with a query type. Within one type, all queries evaluate the same question on a different region of data. For all workloads, we define 5 query types that drill down into distinct combinations of columns.

Lineitem: We generated TPC-H with a scale factor of 50. As lineitem is the main fact table, we use it to schedule predicated



(a) Lineitem (cold cache) **(b) Taxi (cold cache)** **(c) Stocks (after cache warm-up)**

Figure 8: Violin plots of the regions workload. The blue bars report the 25th, 50th, and 75th percentiles, the red dot the mean.

Table 1: Datasets Statistics in MB.

Datasets	TPC-H	Lineitem	Taxi	Stocks
Raw	54,344	37,701	32,473	19,100
Parquet	16,822	10,915	6,858	4,021

queries. Query predicates build upon typical predicates of lineitem and answer questions such as how much revenue was created in the year with low-taxed products.

NYC Taxi: The New York City Taxi dataset includes detailed taxi trips between different regions and locations of the city [22]. Each ride is associated with the duration, price, range, start time, end time, and locations. We drill down on this table to analyze multiple aspects of the data. An example query answers how much tip was accumulated for a region of fares during a certain date range.

Historical Stock Prices: Our second real-world dataset contains historic stock prices of the New York Stock Exchange [39]. With information on open, close, volume, and dates many analytics on stock changes are possible. We execute queries that for example help to determine which stocks had the highest intraday changes in the last year while being traded with high volume.

For a fair comparison, the cache size is given in percentage of the full Parquet remote data. The default value is 20%. Table 1 summarizes statistics on the size of the raw files and the converted and snappy encoded Parquet files.

Crystal supports a wide variety of caches and caching policies:

- No Cache uses vanilla Spark without any code changes
- File Cache (F) caches on a block / file level (traditional cache)
- Requested Region Cache (RR) caches semantic regions eagerly
- Oracle Region Cache (OR) caches semantic regions lazily according to decisions of an oracle

Crystal's caching policies include LRU, LRU-k, DP Knapsack (K_{DP}), and our novel Overlap Greedy Knapsack (K_G). Both knapsack strategies leverage our automatic approximative merging augmentation to find region supersets. For the experiments, we keep a history of 128 regions (see Section 6.8). In the result plots, we combine the cache handle with a caching policy. For example, *RR-LRU*$_2$ denotes a competitor that uses only the requested region cache with LRU-2 as caching policy. Because the number of combinations is large, we focus on the individual strategies and show only two combinations. First, an equal combination of *F-LRU*, *RR-LRU*$_2$, and *OR-K*$_G$ cache, that all get $\frac{1}{3}$ of the caching space. Second, we combine the short-term *RR-LRU*$_2$ and the long-term *OR-K*$_G$ with 95% of the cache denoted to OR. *RR / OR* uses mostly the OR cache because RR is only used to cope with changes that are not yet considered by OR (refresh latency). We propose to use *RR / OR* as it provides superior long-term knowledge because of our overlap-aware *OR-K*$_G$ while being able to adapt quickly to short-term changes with *RR-LRU*$_2$.

6.2 Regions of Query Accesses

This scenario features a regional access pattern which can help reduce future request latencies. Each of the query types spans a region that contains 10 - 15% of the tuples. A query reads between 8 - 13% of a random sample of the region (~1% of the tuples).

The regions workload explores a region by individual queries that overlap in some of the dimensions. The overall union of all the queries within a region represents a large fraction of the region's spanning space. We decide on the region before an individual query is chosen. The regions are accessed in a non-uniform pattern as these span a large percentage of the remote data. Next, one of the 200 pre-computed queries per region is picked randomly. The query parameters are chosen uniformly inside the borders of its type region. The region experiments schedule 400 queries in total. We have made the queries available at [12].

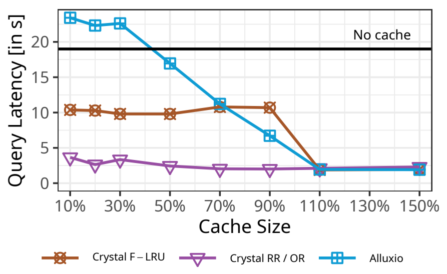
Figure 8 shows this long-term knowledge workload for the lineitem, taxi, and stocks datasets. Note that, the lineitem and taxi experiments run on cold caches. As violin plots give insights into the distribution of the queries, we prefer them over box plots. The shape represents the density of the observations at this value (smoothed by a kernel). Violin plots also encode percentiles and the mean.

Overall, we see significant improvements of the oracle region caches. The greedy knapsack and its *RR/OR* variant with the overlap adaptions outperform the competitors in all three workloads. Because *OR-K*$_G$ benefits most from augmentation, more general regions of interest are found. Especially, at the end of the experiment better caches are used to significantly speed up query processing.

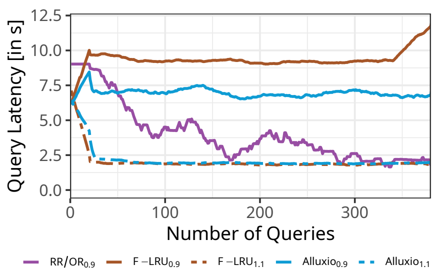
To demonstrate the effectiveness of the better caches, we show the stocks queries after warm-up. At the warm-up phase, the semantic caches (*OR* and *RR-LRU*$_2$) have similar performance. They can capture some of the frequent queries, but fail to generalize. Over time, *OR* learns to cache a better subset. The improvements are shown in Figure 8c. We analyze the cache refresh latency of oracle region caches in Section 6.8.

6.3 Crystal vs. Block Caching

To highlight the performance benefits of Crystal compared to traditional caches, we run the regions workload with different cache sizes. We compare Crystal's *RR / OR* approach with a traditional



(a) Mean lineitem regions performance on varying cache sizes after a warm-up phase.



(b) Lineitem regions mean latency over time with cache sizes 90% (solid) and 110% (dashed).

Figure 9: Comparison against block-based (Alluxio) and file-based caching.

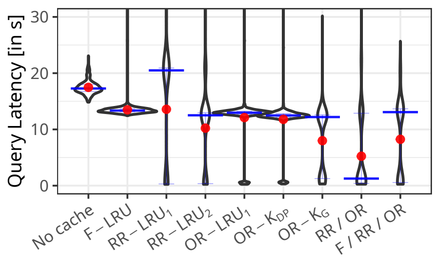


Figure 10: Lineitem regions workload with hot event queries.

file cache (*F-LRU*) and a block-level cache. The block-level cache is represented by Alluxio – which is a widely used accelerator for analytics on disaggregated storage [1, 32, 33].

Figure 9a shows the average lineitem regions performance for different cache values which is denoted by the percentage of the original data. This experiment was conducted after the warm-up phase. Crystal's *RR / OR* is able to learn the set of data needed to accelerate the workload even with limited cache available. Alluxio is initially slower than the vanilla Spark implementation. The larger the cache size, the better Alluxio performs as less blocks have to be evicted. With 110% of the data the maximum performance is reached as all blocks reside on local SSD after the first hit. Only in this extreme scenario, Alluxio is able to match the performance of *RR / OR*. The file based *F-LRU* benefits from an optimized downloading of Parquet files from Azure Blob Storage. Only at very high cache sizes, *F-LRU* is able to improve its performance further.

Even in the high cache size scenarios, *RR / OR* can match the query latencies after warm-up. Figure 9b plots the performance over time compared to the 90% and 110% versions of Alluxio and *F-LRU* with cold caches. Crystal's *RR / OR* is only plotted for 90% (no visible difference to 110%). The traditional caches directly reach the maximum performance whereas *RR / OR* learns the best set of tuples over time. After a sufficiently large number of queries seen, *RR / OR* achieves a similar performance even in the 110% scenario.

6.4 Benefit of RR/OR

In the previous experiments, long-term knowledge was sufficient to solve the scenarios optimally. Real-world workloads often incur a combination of short-term bursts of queries and long-term trends. Similar to a news-breaking event, we reformulate our regions experiment. Regions are still accessed with their respective distribution but additional queries arrive that target different data. These queries access a specific event which first spikes and then falls back to normal. In our scenario, event queries occur up to 8 times (out of 200 queries), before they disappear and another event starts. We schedule $\frac{2}{3}$ regions query and $\frac{1}{3}$ event queries.

The results are shown in Figure 10. Long-term caches (*OR*) fail to catch the events due to their latency constraints. Without long-term knowledge, only event queries can be handled. Although *RR-LRU*$_1$ performs better on short-term queries, the long-term deficits of *RR* caches show that only combined strategies with *RR* and *OR* can overcome both issues. *RR/OR* uses a small short-term cache that catches spikes in reoccurring queries and retains the same performance for long-term trends as *OR* caches.

6.5 Changing Regions

Besides the hot region detection, it is important that caches are refreshed if the workload changes. This experiment takes the regions workload and applies a region change after 400 queries (offset to random value of zipfian) and favors the current region ($\theta = 2$).

Figure 11 depicts the changing regions workload for the lineitem and taxi data. For a better visibility, we only show the semantic base algorithms. The performance is shown as moving average ($n = 40$ queries) over the number of queries. After 400 queries, the hotspot region changes. Both workloads are dominated by *OR-*K_G which is able to reduce runtime while being able to quickly adapt. On the other hand, *OR-*K_{DP} has a long latency until it reaches its optimum again which is easily visible in the lineitem plot. *OR-LRU*$_1$ could adapt fast, however, the overall performance gains are smaller than for *OR-*K_G. The same is true for short-term caches such as *RR-LRU* because they fail to generalize to the semantic regions of interest.

6.6 Crystal's Database Agnostic Design

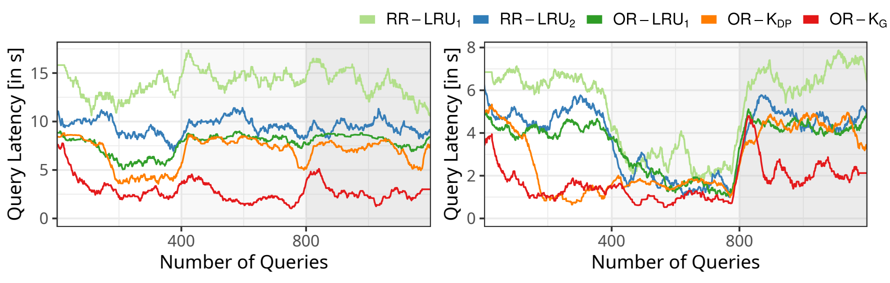
To evaluate our design principle of generality, we show that Crystal can be used as a storage layer for a different DBMS, Greenplum (6.16) [51]. The details of the connector are discussed in Section 5.

Figure 12 depicts the lineitem region workload executed within the Greenplum database. For optimal performance, we configured Greenplum such that it uses as many PostgreSQL workers (segments) as the compute node features hardware threads. Although the performance of Greenplum is slightly worse compared to Spark, the relative benefits of using Crystal stay similar.

6.7 TPC-H

Figure 13a shows the cumulative distribution function for queries generated with the TPC-H Q6 template. We perform the selection according to TPC-H but change the distribution of the shipdate. In many real-world workloads, accesses on date dimensions are heavily skewed. We mimic this behavior by sampling the shipdate from the zipfian distribution. The experiment validates that the *(RR/) OR-*K_G caches capture the hot regions.

Further, we test all 22 queries of TPC-H which are executed twice in this experiment. Due to the construction of TPC-H, only a few queries have atomic predicates on the large tables. Most restrictions come from join conditions which currently cannot be pushed down. Thus, many queries need to get the full table from the remote. Nevertheless, Figure 13b shows that the *OR* approaches are able to improve the mean performance by up to 20%.



(a) Lineitem (b) Taxi

Figure 11: In each third the hot region is changed, i.e. the zipfian is moved to the next region. The plots show the moving average with window size 40 queries.

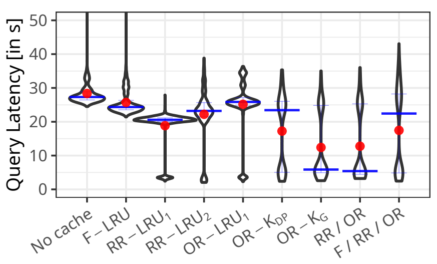
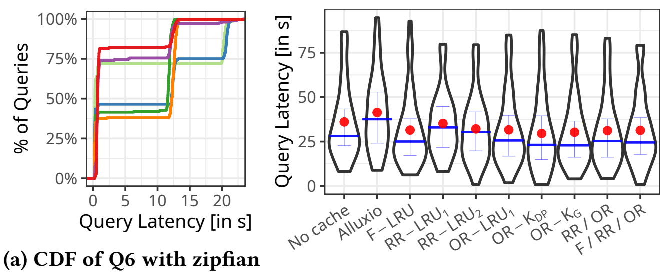


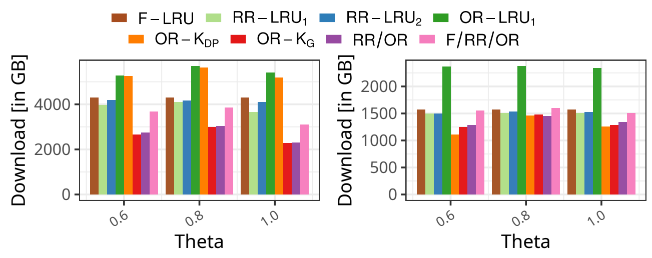
Figure 12: Lineitem regions workload benchmarked on Greenplum.



(a) CDF of Q6 with zipfian distributed date.

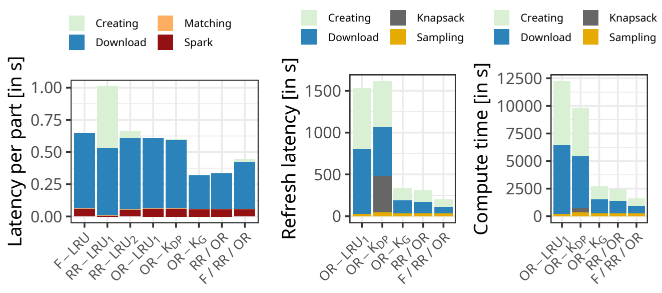
(b) Violin plots of all 22 queries.

Figure 13: Performance on TPC-H.



(a) Regions - Lineitem (b) Regions - Stocks

Figure 15: The total network traffic used by each strategy plotted on a varying theta.



(a) Query latency per Parquet part

(b) Oracle: cache refresh latency

(c) Oracle: sum of compute time

Figure 14: Operation break-down on the online end-to-end and the offline background tasks for lineitem regions.

6.8 Microbenchmarks

To get more insights into the behavior of the caches, we conducted several microbenchmarks. This section evaluates the breakdown of operations, the different parameters, and the network footprint.

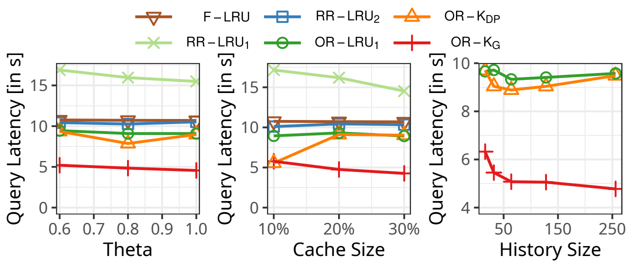
Figure 14 shows the breakdown of the individual operations (mean) during query processing. We split the plots into online (Figure 14a) and offline threads (Figures 14b and 14c). The online thread receives the region request and responses with either cached regions or the raw files downloaded into ramdisk. The largest runtime contributors are the downloading of files and the eager computation. Note that, Figure 14a does not rely on the DBMS to handle the eager file creation. The plot shows that matching does not introduce latency overhead. The background threads are only relevant for oracle region caches. Crystal spawns a low-priority daemon that periodically recomputes the cache. Figure 14b shows the latency until a new cache is computed, Figure 14c sums up the wall-clock of all computing threads. As the threads run at a high nice level, the scheduler does not often consider these threads as soon as load hits the compute node. This results in higher latencies, however, the query processing of the DBMS is not negatively affected by Crystal's background task. The *OR-K_{DP}* has an additional high latency due to the computation of the knapsack. On the other hand, *(RR/) OR-K_G* has low computation effort while caching useful regions. Thus, *(RR/) OR-K_G* downloads and computes less unnecessary files which results in superior background time. *OR-LRU_1* often downloads infrequent regions that will be discarded in the next iteration. Hence, additional and unnecessary computation is introduced.

Figure 15 shows the network footprint for different θ values. *RR/OR-K_G* reduces the network traffic in comparison to our file cache baseline. As more frequent regions are accessed, less traffic is needed. *OR-LRU_1* is prune to downloads of infrequent regions.

Lastly, we vary the used parameters throughout the experimental evaluation in Figure 16. For the regions workload, we show that the algorithms perform similarly for different skew θ of the regions. Nevertheless, the oracle region strategies outperform the baseline in all scenarios. The cache size experiment shows that the oracle algorithms pick the most valuable regions first. Even with a cache that uses only 10% of the data size, good performance is achieved. Smaller history sizes (< 64) decrease the performance as too few queries are collected for predicting the workload. Large history sizes are slower in adopting to new frequent regions.

7 RELATED WORK

The basic idea behind Crystal is to cache and reuse computations across multiple queries. This idea has been explored in a large body of research work including at least four broad lines of research: materialized view, semantic caching, intermediate results reusing, and



(a) Theta. (b) Cache Size. (c) History Size.

Figure 16: Microbenchmarks on lineitem regions.

mid-tier database caching. In general, Crystal differs from previous work in some or all of the following ways: 1) integrating Crystal with a DBMS requires no modification to the DBMS; 2) Crystal focuses on caching views at the storage layer, and can be used across multiple DBMSs; 3) Crystal can automatically choose cached views based on a replacement policy, which takes into account the semantic dependencies among queries. Below, we discuss the key differences between Crystal and previous work in each line of the four aforementioned research areas.

Materialized View. Materialized view is a well-known technique that caches the results of a query as a separate table [20, 44, 45]. However, unlike Crystal, views that need to be cached or materialized are often defined manually by users (e.g., a DBA). Additionally, implementing materialized views in a DBMS is a time-consuming process, requiring advanced algorithms in the query optimizer to decide: 1) if a given query can be evaluated with a materialized view; and 2) if a materialized view needs to be updated when the base table is changed.

Semantic Caching. Semantic caching was first proposed in Postgres [47], and was later extended and improved by a large body of work [11, 14, 17, 29, 30, 42, 43]. This technique also aims to cache the results of queries to accelerate repeated queries. Similarly to Crystal, a semantic cache can automatically decide which views to keep in the cache, within a size budget. This decision is often made based on a cost-based policy that takes several properties of views into consideration such as size, access frequency, materialization cost. However, this approach caches the end results of entire queries, while Crystal caches only the intermediate results of the selection and projection operators of queries. The cached view of an entire query is especially beneficial for repeated queries, but on the other hand decreases the reusability of the cached view, i.e., the chance that this view can be reused by future queries. While most work in this area does not take into account overlap of cached views, some work [14, 17] does explore this opportunity. Dar et al. proposed to split overlapping queries into non-overlapping regions, and thus enable semantic cache to use traditional replacement policies to manage the (non-overlapping) regions [14]. However, this approach could result in a large number of small views, incurring significant overhead to process as we showed in Sec 3.2. Maintaining non-overlapping views is also expensive, as access to an overlapping view may lead to splitting the view and rewriting the cached files. Chunk-based semantic caching [17] was proposed to solve this problem, by chunking the hyper space into a large number of regions that are independent to queries. However, the chunking is pre-defined and thus is static with respect to the query patterns.

Intermediate Results Reusing. Many techniques have also been developed to explore the idea of reusing intermediate results rather than end results of queries. Some of these techniques [49, 50] share the intermediate results across concurrent queries only, and thus impose limitations on the temporal locality of overlapping queries. Other work [19, 25–27, 38, 41] allows intermediate results to be stored so that they can be reused by subsequent queries. Similarly to Crystal, these techniques also use a replacement policy to evict intermediate results when the size limit is reached. However, these techniques require extensive effort to be integrated with a DBMS, whereas integrating Crystal requires only a lightweight database-specific connector. Additionally, a Crystal cache can be used with and share data across multiple DBMSs.

Mid-tier Database Caching. Another area where views can be cached and reused is in the context of multi-tier database architecture, where mid-tier caches [2, 10, 31] are often deployed at the mid-tier application servers to reduce the workload for the backend database servers. As mid-tier caches are not co-located with DBMSs, they usually include a shadow database at the mid-tier servers that mirrors the backend database but without actual content, and rely on materialized views in the shadow database to cache the results of queries. Unlike Crystal, the definition of the cached views in a mid-tier cache needs to be pre-defined manually by users, and it is difficult to change the cached views adaptively.

Finally, many vendors have developed cache solutions for big data systems to keep hot data in fast local storage (e.g., SSDs). Examples include the Databricks Delta Cache [9, 15], the Alluxio [1] analytics accelerator, and the Snowflake Cache Layer [13]. These solutions are based on standard techniques that simply cache files at the page or block level and employ standard replacement policies such as LRU. Compared to these standard approaches, Crystal is also a generic cache layer that can be easily integrated with unmodified big data systems, but has the flexibility to cache data in a more efficient layout (i.e., re-organizing rows based on queries) and format (i.e., Parquet), which speeds up subsequent query processing.

8 CONCLUSION

Cloud analytical databases employ a disaggregated storage model, where the elastic compute layer accesses data on remote cloud storage in columnar formats. Smart caching is important due to the high latency and low bandwidth to remote storage and the limited size of fast local storage. Crystal is a smart cache storage system that colocates with compute and can be used by any unmodified database via data source connector clients. Crystal operates over semantic data regions, and continuously adapts what is cached locally for maximum benefit. Results show that Crystal can significantly improve query latencies on unmodified Spark and Greenplum, while also saving on bandwidth from remote storage.

ACKNOWLEDGMENTS

Dominik Durner has received funding in part from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286). We would also like to thank the anonymous reviewers and Donald Kossmann for their feedback and support of this work.

REFERENCES

[1] Alluxio. 2021. Alluxio - Data Orchestration for the Cloud. https://www.alluxio.io/. accessed: 2021-07-16.

[2] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2003. Cache Tables: Paving the Way for an Adaptive Database Cache. In *VLDB*. 718–729.

[3] Amazon. 2020. Amazon S3 Cloud Storage. https://aws.amazon.com/s3/. accessed: 2021-07-16.

[4] Amazon. 2020. Amazon S3 Select. https://aws.amazon.com/blogs/aws/s3-glacier-select/. accessed: 2021-07-16.

[5] Apache. 2021. Apache Arrow. https://arrow.apache.org/. accessed: 2021-02-17.

[6] Apache. 2021. Apache Avro. https://avro.apache.org/. accessed: 2021-02-17.

[7] Apache. 2021. Apache Parquet. https://parquet.apache.org/. accessed: 2021-02-17.

[8] Apache. 2021. Gandiva for Apache Arrow. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/. accessed: 2021-02-17.

[9] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. 2020. Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores. *PVLDB* 13, 12 (2020), 3411–3424.

[10] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. 2004. Adaptive Database Caching with DBCache. *IEEE Data Engineering Bulletin* 27, 2 (2004), 11–18.

[11] Chung-Min Chen and Nick Roussopoulos. 1994. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In *EDBT*. 323–336.

[12] Crystal. 2021. Regions workload queries. https://aka.ms/crystal-queries. accessed: 2021-07-16.

[13] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[14] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *VLDB*. 330–341.

[15] Databricks. 2021. Delta Cache. https://docs.databricks.com/delta/optimizations/delta-cache.html. accessed: 2021-02-17.

[16] Databricks. 2021. Introduction to Data Lakes. https://databricks.com/discover/data-lakes/introduction. accessed: 2021-02-17.

[17] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching Multidimensional Queries Using Chunks. In *SIGMOD*. 259–270.

[18] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads. *PVLDB* 14, 2 (2020), 74–86.

[19] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. 2017. Revisiting Reuse in Main Memory Database Systems. In *SIGMOD*. 1275–1289.

[20] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A practical, scalable solution. In *SIGMOD*. 331–342.

[21] Goetz Graefe. 2009. Fast loads and fast queries. In *DaWaK*. 111–124.

[22] Evan Hallmark. 2019. Daily Historical Stock Prices. https://www.kaggle.com/ehallmar/daily-historical-stock-prices-1970-2018. accessed: 2021-01-17.

[23] Karl Holub. 2021. The Overlapped Hyperrectangle Problem. https://kholub.com/projects/overlapped_hyperrectangles.html. accessed: 2021-02-17.

[24] Oscar H Ibarra and Chul E Kim. 1975. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)* 22, 4 (1975), 463–468.

[25] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. 2009. An architecture for recycling intermediates in a column-store. In *SIGMOD*. 309–320.

[26] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB* 11, 7 (2018), 800–812.

[27] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation Reuse in Analytics Job Service at Microsoft. In *SIGMOD*. 191–203.

[28] Srikanth Kandula, Laurel Orr, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates through Joins in Big-Data Clusters. *PVLDB* 13, 3 (2019), 252–265.

[29] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. 2000. Cache investment: integrating query optimization and distributed data placement. *TODS* 25, 4 (2000), 517–558.

[30] Yannis Kotidis and Nick Roussopoulos. 1999. DynaMat: A Dynamic View Management System for Data Warehouses. In *SIGMOD*. 371–382.

[31] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. 2004. MTCache: Transparent Mid-Tier Database Caching in SQL Server. In *ICDE*. 177–188.

[32] Haoyuan Li. 2018. *Alluxio: A virtual distributed file system*. Ph.D. Dissertation. UC Berkeley.

[33] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. 2014. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *SoCC*. 6:1–6:15.

[34] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *SIGMOD*. 2530–2542.

[35] Microsoft. 2020. Azure Data Lake Storage Query Acceleration. https://docs.microsoft.com/en-us/azure/storage/blobs/data-lake-storage-query-acceleration. accessed: 2021-07-16.

[36] Microsoft. 2020. Azure Storage - Secure cloud storage. https://azure.microsoft.com/en-us/services/storage/. accessed: 2021-07-16.

[37] Microsoft. 2021. Azure Storage C++ Client Library (Lite). https://github.com/Azure/azure-storage-cpplite. accessed: 2021-07-16.

[38] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. 2013. Recycling in pipelined query evaluation. In *ICDE*. 338–349.

[39] City of New York. 2021. New York City TLC Trip Record Data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page. accessed: 2021-03-01.

[40] Oracle. 2021. Database Data Warehousing Guide: Using Zone Maps. https://docs.oracle.com/database/121/DWHSG/zone_maps.htm. accessed: 2021-07-16.

[41] Luis Leopoldo Perez and Christopher M. Jermaine. 2014. History-aware query optimization with materialized intermediate views. In *ICDE*. 520–531.

[42] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. WATCHMAN : A Data Warehouse Intelligent Cache Manager. In *VLDB*. 51–62.

[43] Junho Shim, Peter Scheuermann, and Radek Vingralek. 1999. Dynamic Caching of Query Results for Decision Support Systems. In *SSDBM*. 254–263.

[44] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. 1998. Materialized View Selection for Multidimensional Datasets. In *VLDB*. 488–499.

[45] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. 1996. Answering Queries with Aggregation Using Views. In *VLDB*. 318–329.

[46] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. 553–564.

[47] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. 1990. On Rules, Procedures, Caching and Views in Data Base Systems. In *SIGMOD*. 281–290.

[48] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. 2014. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*. 1115–1126.

[49] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. 2001. Cache-on-Demand: Recycling with Certainty. In *ICDE*. 633–640.

[50] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2019. Intermittent Query Processing. *PVLDB* 12, 11 (2019), 1427–1441.

[51] VMware. 2021. Greenplum Database - Massively Parallel Postgres for Analytics. https://www.greenplum.org/. accessed: 2021-07-16.

[52] VMware. 2021. Greenplum Platform Extension Framework (PXF). https://docs.greenplum.org/6-8/pxf/overview_pxf.html. accessed: 2021-07-16.

PAPER P3

JSON Tiles: Fast Analytics on Semi-Structured Data

Synopsis. In addition to structured (relational) formats, semi-structured data formats are commonly used. These formats combine both the data content and the structural information. Thus, no upfront schema definition is required, which reduces API complexity. Developers often prefer these formats for their flexibility and ease of data transfer. JSON is the most common type of semi-structured formats and can represent arbitrarily complex hierarchical data. Because of the widespread use as a transfer format, large JSON datasets are accumulated by logging software or collected from publicly facing APIs. Analytics on such datasets is valuable, but the interleaving of data and schema introduces overhead in scanning and extracting valuable information.

The goals of processing semi-structured data are to achieve high access performance, maintain robustness regarding heterogeneous data, and enable query optimization. The traditional approach in database systems to store complete documents (plain-text or binary-optimized) involves access into every document for information retrieval, which is significantly slower than reading data from columnar storage. Earlier research investigates raw JSON parsing at rates exceeding one GiB per second per core. However, querying raw collections remains CPU intensive as every query parses the entire dataset, and no statistics for query optimization are collected. Although JSON is capable of representing arbitrary data, documents from the same source exhibit structural overlap. Previous work explores this structural similarity by collecting globally common keys and extracting them into columns but fails to provide robustness to changing or even heterogeneous data.

This paper introduces *JSON tiles*, a collection of algorithms for high-performance JSON processing. *JSON tiles* automatically identifies the implicit structure of a fine-grained collection of documents (tile) and extracts data into column chunks, which facilitate fast query processing. By using a finer granularity, changes in the document structure over time are automatically accounted for. Additional reordering of documents between different tiles allows *JSON tiles* to materialize JSON documents from heterogeneous data sources, providing strong robustness guarantees. During insertion, statistics are collected for each tile, which are then aggregated to table-wide statistics used for join ordering. Results on different JSON collections show that *JSON tiles* is the first approach to address all three defined goals of processing semi-structured data.

Contributions. Dominik Durner contributed substantially to the content of the paper, in particular concerning the development of the proposed ideas, the implementation of the system, the evaluation, and authoring substantial parts of the paper.

Reference. Dominik Durner, Viktor Leis, and Thomas Neumann. "JSON Tiles: Fast Analytics on Semi-Structured Data". In: *SIGMOD*. ACM, 2021, pp. 445–458

DOI. `https://doi.org/10.1145/3448016.3452809`

Miscellaneous. This paper received an Honorable Mention at SIGMOD 2021[1].

[1]`https://2021.sigmod.org/sigmod_best_papers.shtml`

JSON Tiles: Fast Analytics on Semi-Structured Data

Dominik Durner
Technische Universität München
dominik.durner@tum.de

Viktor Leis
Friedrich-Schiller-Universität Jena
viktor.leis@uni-jena.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Developers often prefer flexibility over upfront schema design, making semi-structured data formats such as JSON increasingly popular. Large amounts of JSON data are therefore stored and analyzed by relational database systems. In existing systems, however, JSON's lack of a fixed schema results in slow analytics. In this paper, we present *JSON tiles*, which, without losing the flexibility of JSON, enables relational systems to perform analytics on JSON data at native speed. JSON tiles automatically detects the most important keys and extracts them transparently – often achieving scan performance similar to columnar storage. At the same time, JSON tiles is capable of handling heterogeneous and changing data. Furthermore, we automatically collect statistics that enable the query optimizer to find good execution plans. Our experimental evaluation compares against state-of-the-art systems and research proposals and shows that our approach is both robust and efficient.

CCS CONCEPTS

• **Information systems → Semi-structured data**; **Data layout**; *Online analytical processing engines*; *Database query processing*.

KEYWORDS

Semi-structured data; JSON; JSONB; Storage; Analytics; OLAP; Scan

ACM Reference Format:
Dominik Durner, Viktor Leis, and Thomas Neumann. 2021. JSON Tiles: Fast Analytics on Semi-Structured Data. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event , China.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3448016.3452809

1 INTRODUCTION

A plethora of data is created every day and forecasts show that data volume will rapidly increase in the next years [43]. Much of this data is semi-structured, i.e., it combines the data content and the schema. The most common semi-structured format today is the JavaScript Object Notation (JSON), a human-readable plain text storage format that allows representing arbitrarily-complex hierarchies. Large JSON data sets are, for example, accumulated when logging software system events or collecting data through public web APIs, such as the JSON APIs of Facebook [24], Twitter [60], and Yelp [64]. Public JSON data sets are also used to enrich proprietary data that is stored in relational systems. Analytics on large JSON data is valuable but expensive. Specialized tools for log file analysis, such as Splunk [55] exist, but lack the flexibility and functionality of general-purpose data management systems.

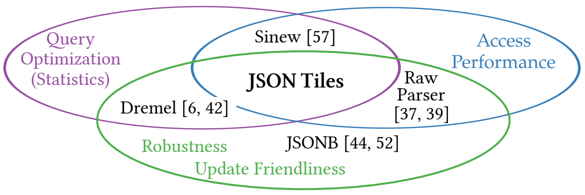


Figure 1: Classification of existing work.

To speed up analytical processing of JSON data, a number of approaches have been proposed. Figure 1 classifies them with respect to access performance, robustness to heterogeneous data, and query optimization. SIMD-JSON [37] and Mison [39] allow parsing JSON with up to one GB/s per core. However, querying documents remains expensive because access to a single field requires a full parse over the data. Relational database systems store each JSON object as a string or use a per-object binary representation [52]. Both approaches are inefficient for analytical queries in comparison with relational column stores. Sinew [57] therefore extracts complete columns to speed up accesses. However, it can only produce good columnar extracts if the data mostly consists of the same static document structure. Sinew does not handle changing or heterogeneous data well and updates are expensive because new document structures change the global frequency of common keys. Reassembling shredded documents with different structures at a record level, as performed with Dremel [42] and implemented in Apache Parquet [6], results in additional work during query execution: many different optional fields have to be handled while evaluating the access automata. Processing Parquet files is CPU-bound even for purely relational files without optional fields [14].

This paper presents *JSON tiles*, a collection of algorithms and techniques that enable high-performance analytics on JSON data. The key idea behind JSON tiles is to automatically detect the implicit common structure across a collection of objects. Using this structural information, we infer types, materialize frequently occurring keys as relational columns, and collect query optimizer statistics – enabling performance close to that of native relational column stores. Infrequent keys and heterogeneous (outlier) objects are stored in an optimized binary format that allows fast access to individual keys. All these techniques are automatic and transparent, enabling fast analytics on JSON data without sacrificing the flexibility of the format.

We integrated JSON tiles into our RDBMS Umbra, which provides SQL, columnar storage, a fast query engine, and a cost-based query optimizer [34, 47]. Using JSON tiles, we leverage these mature technologies, which have been developed in a relational setting, for analytics on JSON data. This paper describes the deep integration necessary and may therefore serve as a blueprint showing how to extend existing systems with high-performance JSON support.

2 DESIGN OVERVIEW

In principle, JSON objects can have very complex structures and each object can have a different implicit schema. However, in practice, JSON data is often machine-generated and has a fairly rigid and predictable structure. The key idea of our approach is to detect and leverage this implicit structure to speed up query processing.

2.1 Challenges

We first define three design goals before outlining how JSON tiles achieves these goals.

Access Performance: Accessing attributes of JSON documents requires document traversal. This traversal introduces a large overhead as every tuple requires a new lookup and all values are untyped. Accesses of relational columns, on the other hand, are cheap – in particular in column stores. This creates a big performance gap between JSON and relational attribute accesses. JSON tiles gains insights during data loading such that data can be stored in a columnar representation. This enables fast scans of JSON data.

Query Optimization: Traditional RDBMS collect statistics (such as histograms and distinct counts) on each column. As each JSON document is stored as one opaque tuple, the statistics are created based on full (textual) JSON representation. For example, this would likely result in the number of distinct values corresponding to the table's cardinality. However, scan and join conditions usually access individual keys, and such statistics do not help in estimating selectivities.

For meaningful statistics, each document must be traversed and statistics on individual keys must be gathered. For example, join ordering uses distinct values of attributes to estimate the join cardinality. Without individual statistics, the optimizer relies on imprecise estimates. Thus, the query plan can be very inefficient [38] (e.g., because a bad join order is selected).

JSON tiles exploits the structural information gathered during loading to maintain data statistics. As the number of keys is unbounded, JSON tiles stores statistics on the frequent keys for precise estimates. This enables complex multi-table queries without having to manually transform the data to a relational schema first.

Robustness on Heterogeneous Data: The convenience of putting arbitrary documents into the database is often the primary reason for choosing semi-structured formats. Although the structure of objects is not arbitrary in practice, many data sets contain heterogeneous document types. Consequently, the storage engine needs to adapt to heterogeneous documents, changes of fields, and previously unseen data. For example, documents tend to grow over time as more and more fields are added to the original document type. Another important use case is the combination of log data from multiple sources. It is infeasible to define a global schema upfront for analytics on combined log data. As the analytics on JSON data was expensive in a general DBMS, log data is often analyzed by specialized providers such as Splunk [55].

JSON tiles handles different document types and copes with outliers through local computations. Further reordering helps in randomized insertions of heterogeneous documents.

2.2 Leveraging Implicit Document Structure

We explain the key ideas of JSON tiles using a running example that consists of real-world JSON documents from Twitter's public API. Figure 2 shows a simplified example of 8 JSON documents representing information about tweets. Every document consists of an identifier, the tweet text, a create field, and a user object. As is common in many real-world data sets, the attributes of tweets changed over time. For example, Twitter introduced the famous hashtags after user feedback in 2007 and further attributes like reply (2007), retweet (2009), geo-tags (2010) were added over time [61].

Observations: As the example illustrates, the JSON documents in a collection often have the same set of keys and, therefore, have a similar implicit schema. Furthermore, the values for a key have matching types as well. In the example, the *identifier* attribute stores integers and the tweets (not shown in the example) would be textual strings. Another interesting type can be derived from *create* key. Although it is represented as a string because JSON does not specify a date data type, a query will most likely use it as date object. These observations lead to the insight that real-world semi-structured databases often effectively contain relational information.

Consequently, using the key structure and observed values, we can materialize the common structures as typed relational columns. However, detecting a single global relational schema, as proposed by Sinew [57], may be problematic. Simply materializing all keys as columns may lead to many null entries. Using some frequency cutoff, e.g., only extracting a particular column if at least 80% of all documents have that key, may prevent relevant columns from being extracted. In our example, the *replies* and *geodata* cannot be extracted by a global detection algorithm that extracts all keys that are represented by more than $\frac{2}{3}$ of all documents.

Our approach therefore breaks the input documents into multiple chunks – which we call *JSON tiles.* We search for local substructures within the smaller chunks to find more common patterns. We also automatically infer the data types and assume that values that look like a certain type will most likely be used as such. The small granularity of JSON tiles also enables parallelizing bulk loading as tiles can be constructed largely independently.

Outlier Handling: As JSON tiles collects document structures locally, it is likely that fewer document structures are observed in comparison to a global collection of structures. This already reduces the number of potentially materialized but unused columns, and thereby the number of null entries from absent fields. Because tiles are restricted in the number of tuples, a higher percentage of potential outliers, such as the missing geo-info, can be accepted. Hence, JSON tiles does not miss frequent keys and is able to adapt to changes of data objects and arrays, which results in a more robust system. New keys are added to the materialized parts, whereas removed keys are not extracted in future tiles.

Column Extraction: Because data is materialized into a columnar format, no semi-structured access computations are necessary. The cost of accessing a column chunk is amortized by the number of tuples scanned. Therefore, our approach achieves high analytical columnar scan performance while being robust to heterogeneous data objects or combined log data documents from different sources. In the Twitter example, our approach is able to extract replies and geodata into column chunks of the second tile.

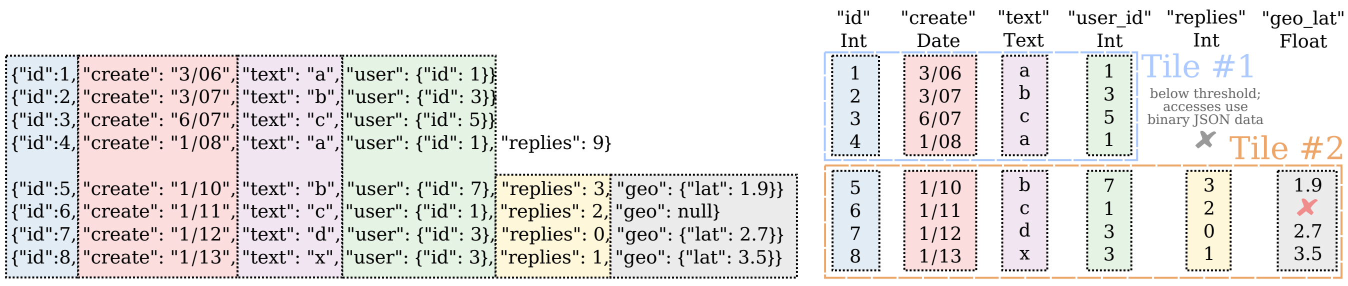

{"id":1, "create": "3/06", "text": "a", "user": {"id": 1}}
{"id":2, "create": "3/07", "text": "b", "user": {"id": 3}}
{"id":3, "create": "6/07", "text": "c", "user": {"id": 5}}
{"id":4, "create": "1/08", "text": "a", "user": {"id": 1}, "replies": 9}
{"id":5, "create": "1/10", "text": "b", "user": {"id": 7}, "replies": 3, "geo": {"lat": 1.9}}
{"id":6, "create": "1/11", "text": "c", "user": {"id": 1}, "replies": 2, "geo": null}
{"id":7, "create": "1/12", "text": "d", "user": {"id": 3}, "replies": 0, "geo": {"lat": 2.7}}
{"id":8, "create": "1/13", "text": "x", "user": {"id": 3}, "replies": 1, "geo": {"lat": 3.5}}
"id" Int
"create" Date
"text" Text
"user_id" Int
"replies" Int
"geo_lat" Float
Tile #1
below threshold; accesses use binary JSON data
Tile #2

Figure 2: Twitter Tweet JSON data extracted into two JSON tiles.

Statistics: As we collect the smaller JSON tiles, we trace whether keys have been materialized and compute query optimizer statistics. We store the statistics information in each JSON tile. The local statistics are then propagated to generate table statistics such that they give details about the input data. This information is used to find efficient query plans that minimize intermediate join results.

3 EXTRACTION

This section presents the fundamental ideas behind JSON tiles and the algorithms for constructing them.

3.1 JSON Tiles

Previous work by Tahara et al. [57] observed that documents in real-world data sets often have similar structure and they therefore propose extracting a single schema globally. However, such a global approach is not robust with respect to heterogeneous or changing data. Depending on the chosen extraction threshold, many keys will either fall below the threshold or the resulting relation will have many attributes with mostly null values. In both cases, performance will not be optimal for heterogeneous data sets.

We therefore propose to detect the implicit document structure at a fine granularity (hundreds or thousands of documents rather than globally). We split the input data into disjunct *JSON tiles*, for each of which we detect a local schema. This approach naturally exploits the spatial locality contained in many data sets and finds a sweet spot between fast scan performance and the reduction of uncommon patterns. Our experiments show that a tile size of 2^{10}-2^{12} tuples works well across many workloads.

In the following, we show the extraction steps for JSON tiles. Tile #2 of Figure 2 acts as our running example. The tile size of the tweet data is 4 tuples and we use an extraction threshold of 60%.

(1) Collect all key paths for each tuple: A key path is the path of nested objects and arrays followed to the actual key-value pair. For an easier notation, we will use only the first letter of each key and encode the nesting with '_'. For instance, the tuple with id 5 has the key paths { i , c , t , u_i , r , g_l }. Tuples 7 and 8 have the same key paths, whereas tuple 6 lacks g_l .

(2) Use the collected key paths as input for frequent itemset mining: An itemset is frequent if it exceeds the extraction threshold. The extraction threshold is the frequency count, which counts how many tuples contain this itemset, divided by the overall number of tuples. The itemset miner finds subsets within the collected key paths that are frequent. In our example, the miner finds two frequent maximum subsets and their frequency: ({ i , c , t , u_i , r }, 4) and ({ i , c , t , u_i , r , g_l }, 3). They are maximum itemsets as each further subset of ({ i , c , t , u_i , r }, 4) has the same frequency. All details and constraints of the itemset mining algorithm are explained in Section 3.3.

(3) Extract the union of the maximum itemsets: JSON tiles iterates over the found itemsets and extracts the key paths as materialized relational columns. All key paths are materialized from the first maximum subset. As the first and second maximum subset overlap, only g_l is additionally materialized. This results in the final extraction of { i , c , t , u_i , r , g_l } for Tile #2.

{"id":1, "date": "1/11", type: "story", "score": 3, "desc": 2, "title": "...", "url": "..."}
{"id":2, "date": "1/12", type: "poll",   "score": 5, "desc": 2, "title": "..."}
{"id":3, "date": "1/13", type: "pollop", "score": 6, "poll": 2, "title": "..."}
{"id":4, "date": "1/14", type: "story", "score": 1, "desc": 1, "title": "...", "url": "..."}
{"id":5, "date": "1/15", type: "comment", "parent": 4, "text": "..."}
{"id":6, "date": "1/16", type: "comment", "parent": 1, "text": "..."}
{"id":7, "date": "1/17", type: "pollop", "score": 3, "poll": 2, "title": "..."}
{"id":8, "date": "1/18", type: "comment", "parent": 1, "text": "..."}

Figure 3: News items [28] with different document types.

3.2 Tile Partitions and Tuple Reordering

A new tile is created whenever the number of newly-inserted tuples reaches the tile size. Consequently, the content of JSON tiles depends on the insertion order. For many applications, the insertion order already provides strong spatial locality and therefore high-quality JSON tiles. For instance, adding fields over time, as in the Twitter example, results in almost perfect tiles. However, workloads like the one shown in Figure 3, where each document is of a different type, have little spatial locality. Even fine-granular tiles would result in poor scan performance. In the following, we describe an approach that solves this issue by reordering tuples between neighboring tiles.

The goal of the reordering algorithm is to find frequent itemsets across multiple tiles. The tuples are then reordered such that the same frequent itemsets are clustered in a single tile. The neighboring tiles grouped together for reordering are denoted as a *partition*.

Reordering is illustrated in Figure 4, which uses a tile size of 5 tuples and shows 12 tiles that are split into partitions of size 4. Each tile mines frequent itemsets with a reduced threshold. In the example, every patterned rectangle represents a tuple and the pattern denotes the frequent itemset that describes the tuple best. If we assume that all of the different patterns have no key paths in common, no materialization would be possible without reordering. JSON tiles clusters tuples into the tiles such that every itemset cluster satisfies the original threshold. The tuples are then distributed accordingly.

Once the tile redistribution has been performed, most tiles are perfectly extractable in the example. Each tile has a frequent structure that is over the extraction threshold. However, some tiles contain tuples that cannot be materialized. In contrast, before reordering none of the patterns exceeded the threshold in any tile. Our experiments on multiple workloads show that a partition size of 8 tiles yields good results.

The full algorithm for reordering proceeds as follows:

(1) The frequent itemsets of each individual JSON tile are mined. As these itemsets are used for reordering, the threshold for being frequent is reduced to $\frac{\text{threshold}}{\text{partition size}}$.

(2) The itemsets of all tiles within one partition are exchanged. Itemsets with a frequency of more than threshold $*$ tile size survive.

(3) Every tuple in the partition is matched to the frequent itemset that describes it best. The algorithm picks the largest itemset that has the most items in common with the tuple. As the itemset mining needs to be limited (see Section 3.3), ties need to be resolved such that every tuple that encounters this tie will match the same itemset. For example, our JSON tiles implementation resolves ties by minimizing the sum of item ids for equal matches.

(4) While matching the tuples, a hash table aggregates the count of the itemsets for both the individual tiles and the partition. As each tuple is only matched to one itemset, the tuples are simply put into individual tiles such that the original extraction threshold is reached (if possible). This mapping is computed in a greedy fashion.

(5) With the current count of tuples matching the itemsets in the tile as well as those required to satisfy the mapping, the swap positions between tiles are computed. The algorithm iterates over all tuples. If the tuple is needed in the current tile, no swapping is performed. Otherwise, the tuple is swapped with another tuple that matches the need for this tile. For example, a tuple is of itemset type green and in tile 1. Further, tile 1 needs to be filled with type purple and tile 2 with type green. First, tile 2 will be searched for a matching tuple of type purple as this would benefit both tiles. If tile 2 does not have any tuple of type purple, which is directly visible from the aggregate map, the remaining tiles are searched.

(6) The last step simply computes the itemsets with the original threshold of the reordered tiles to find the final extraction columns. Even if tuples belong to different itemsets, they can share key paths.

As is indicated by Figure 4, the tile partitioning parallelizes well on larger data sets. Each thread is dedicated to a disjoint subset of the data (partition). No interaction is needed as the information is disjoint between different threads. During tile creation, no issues for concurrent scans arise, as the tile is visible to scanners only once it is fully created. Only if tuples are currently being swapped, concurrent readers need to block until the swapping is finished.

3.3 Frequent Itemset Mining

JSON tiles uses frequent structures to materialize columns and redistribute tuples between them. These structures are found by gathering information on all available key paths. The frequent itemset miner determines which items are common and therefore materialized. The knowledge of itemsets helps to find the best frequent representation so that similar tuples can be redistributed. Furthermore, reordering within a tile improves compression in systems that support run-length encoding.

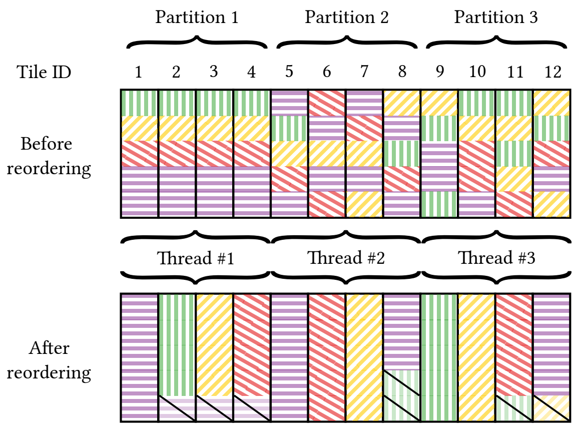


Figure 4: Reordering with partition size 4. Each tile has 5 tuples (vertical) and the extraction threshold is 60%.

To compute frequent itemsets, we rely on an efficient implementation of the *FPGrowth* algorithm [29]. In comparison to the classic *Apriori* [1] variant, *FPGrowth* does not need to generate candidate sets. *FPGrowth* creates a tree of frequent items and recursively iterates over the tree to generate output sets. We collect all keys from the documents and store them dictionary encoded. Dictionaries are created for every JSON tile and are used as the database to mine.

Unfortunately, the complexity of the result is a major problem of itemset mining. Since in the worst case the number of frequent itemsets is equal to the cardinality of the powerset of frequent items, we need to restrict the number of computed itemsets. Otherwise, itemset mining would be prohibitively expensive for tile creation.

As we only want to gracefully decrease precision, the algorithm computes itemsets until a budget is reached. Smaller itemsets are computed first as these are needed for larger ones. All frequent items are used to find potential itemsets that can be used for extraction. However, the number of elements (k) in the potential sets needs to be restricted. We denote a budget u as the upper bound of itemsets.

$$\sum_{i=1}^{k} \binom{n}{i} \leq u' \leq 2^n - 1, \text{ with } u' \leq u \tag{1}$$

We choose all 1 to k subsets of an n-ary set, resulting in the summation of the binomials. We compute k such that the number of generated subsets is limited to u', which is always smaller than 2^n and u. Because k is dependent on the depth of the recursive mining of conditional pattern trees generated by FPGrowth, we bound the operations. As the recursion depth is restricted, the system is not overloaded during JSON tile materialization.

3.4 Value Types and Key Paths

In JSON, multiple values for the same key do not necessarily have the same primitive JSON type, e.g., some values are integer and some are float. If we decide to extract that key, we have to decide which data type to assign to the extracted column. At the same time, it must be ensured that the original type information is not lost and that JSON semantics is maintained.

To solve this problem, the tile extraction algorithm combines the key path with the primitive JSON type, i.e., each itemset entry is actually a pair and two key paths only match if their value types match as well. This way, if several options are available, extraction will chose the most common type. Assume, for example, that the same key path contains integers as well as floats, and that the integers are extracted. This means that the float values cannot be stored in an extracted form and have to be stored in the binary JSON representation (cf. Section 5). On access, for example when summing up all values, we therefore traverse the binary representation when the extracted column value is null. This approach maintains JSON semantics for outliers, while providing fast scan performance for the majority of values.

3.5 Nested Objects and Arrays in JSON Tiles

A major feature of JSON is its capability to nest objects arbitrarily. Our extraction algorithm handles nesting by encoding it into the key path. During extraction, JSON tiles thereby do not have to distinguish between nested and non-nested objects. During the key path retrieval, the nesting level is computed as well as the followed keys. In the Twitter example (Figure 2), the nested key *lat* is extracted and encoded with its nested path (*geo*→*lat*).

Accessing nested column extracts require some care. For example, the access to `'key'`→`'nestedKey'` could first extract the object `key` and then use a regular JSON lookup or access `nestedKey` directly if available. Usually, a direct access to `nestedKey` is preferred. However, the database needs to know whether an access to `key` is needed as other expressions could use `key` as well.

To overcome this issue, JSON tiles recognizes during the scan operation whether the other levels of the key path are needed. We count how often each key is used as multiple expressions are able to share the same paths. If the path is used exclusively by one expression and the `nestedKey` is materialized, the intermediate access is removed. The final lookup is a simple access of this extracted key.

Another interesting challenge arises from heavily nested arrays. If the number of elements in an array is similar in all documents of a tile, JSON tiles is able to materialize all frequent elements. However, if the number varies, JSON tiles materializes only the leading elements that are frequent across all documents. For example, if every document contains an array with x elements but some documents have $x + c$ array elements, only the first x elements are extracted.

This issue can be addressed by combining our approach with prior work. Deutsch et al. [19] distinguish between high-cardinality arrays and small-set arrays. The issue described only arises with high-cardinality arrays that contain many nested objects and differ significantly in the element count. Related work suggests extracting high-cardinality arrays into separate tables. The details on the orthogonal problem of detecting high-cardinality arrays are discussed in [19, 54]. After these arrays are determined, our JSON tiles extraction algorithm is used to automatically materialize additional tables from the detected arrays. In Section 6.3, we evaluate a combined approach in the presence of high-cardinality arrays.

4 INTEGRATION

JSON tiles touches many components of the DBMS. This section explains the adaptions that are necessary for a seamless integration.

4.1 Accessing JSON Attributes

In relational database systems, JSON data is usually stored in a single column of a table. Each value of this kind of JSON column holds a full JSON document. They are stored as JSON strings, which is a verified human-readable textual representation. JSON columns do not have any additional information on the structure of the contained documents. Some systems use a per-document optimized binary JSON format. This improves access performance by storing data in a binary representation that has minimal parsing overhead.

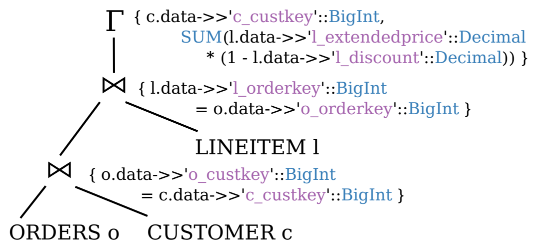

Γ { c.data->>'c_custkey'::BigInt,
SUM(l.data->>'l_extendedprice'::Decimal
* (1 - l.data->>'l_discount'::Decimal)) }
⋈ { l.data->>'l_orderkey'::BigInt
= o.data->>'o_orderkey'::BigInt }
LINEITEM l
⋈ { o.data->>'o_custkey'::BigInt
= c.data->>'c_custkey'::BigInt }
ORDERS o
CUSTOMER c

Figure 5: Join tree of simplified TPC-H query 10 before access expression push down.

The most basic operation on JSON data is attribute access. In the examples throughout this paper, we use PostgreSQL-style access operators: the access as JSON type (->) and the access as Text (->>) expressions evaluate JSON queries [51]. These expressions are needed since the information is stored in nested objects and arrays. They return the value to the key (object) or slot (array).

For example, `{"id":0, "name":"JSON"}` is a JSON object that holds two keys with one integer and one string value. Assuming that the user wants to access the id field, it can be requested as a value of type JSON with `object->'id'` resulting in `{0}`. Note that the result type is not the integer itself. The access as JSON object function is necessary to access nested objects since access expressions can only be evaluated on JSON documents. The other option is to access the element as text with `object->>'id'`, which returns the Text `"0"`. Because the user usually intends to access the pure integer value, a cast from the string representation is needed. The expression is therefore rewritten to `object->>'id'::Int`, which finally outputs the Integer `0`. Umbra follows the PostgreSQL semantics of returning null if the requested key or any parent key is not present.

4.2 Push Down of Access Expressions

To utilize the scan performance of JSON tiles, changes to the query plan are necessary. The scan operator needs information on the keys that are accessed to decide whether extracts of tiles can be used. Previous work showed that the push down of accesses into the scan operator is crucial to heterogeneous data formats [33]. In the following, we describe the push down of JSON accesses and explain the steps to integrate JSON tiles into the query plan.

Figure 5 shows the query plan of a simplified version of TPC-H query 10 that uses JSON. In this example, the data is stored in a single JSON column (data). Each row is transformed into a JSON document such that every column name works as the key in the JSON objects. Because the operators above the scan need the JSON string for expression evaluation, each table scan operator has to produce the whole JSON string. Using the whole string when only parts are needed is inefficient.

As JSON tiles relies on the usage of extracted columns, the table scan operator needs to know which parts of the JSON data are accessed. If access expressions are evaluated further up the query plan, the table scan needs to provide the raw JSON data and cannot utilize the materialized columns of JSON tiles. Thus, the access expression evaluation has to be pushed down into the scan operator.

We use placeholders for expressions and hand the computation of the access expressions over to the table scan operator. The result of an expression is then available at this location and directly usable by a parent operator. If a column extract of an expression is available, the data is read from the extract and the placeholder simply points to the materialized data. Otherwise, the raw JSON value is accessed.

4.3 Cast Rewriting

Because the return type of JSON accesses is Text, it is important to also push down the cast type information to the scan operator. Otherwise, the materialized types need to be transformed to Text first and later have to be transformed back to the cast requested type. This introduces a large query runtime overhead.

During the optimization phase of the RDBMS, cast rewriting reduces this overhead. The RDBMS checks whether the input expression of the cast is an access as Text lookup. If so, the cast result type determines which specialized access expression is used.

The RDBMS implements optimized access expressions for every data type that is defined for JSON documents (Section 5.1). Since these types are also used for the JSON tiles extraction, it is beneficial to rewrite these expensive casts. If the original type matches the cast result, we simply delete the cast operator and return the evaluated access expression directly. Otherwise, we reduce cast overhead as a better cast option is chosen. For example, a BigInt stored element key with the lookup `x->>'key'::Float` is rewritten to a BigInt lookup followed by a cheap cast to Float.

4.4 Storing the JSON Tiles Header

Because JSON tiles detects frequent document structures from the input seen locally, the extracted columns vary between different tiles. Thus, each tile needs its own header describing its seen and materialized data. For accessing materialized data, JSON tiles needs to store the extracted key paths and the corresponding value types. Since tiles vary in data and size, they are not directly stored in the fixed-sized part of the relation but in the variable-sized data. Only the pointer to the header is stored in the relation to map from tuples to the corresponding tile. Offsets into the variable-size data remain static as we either append the memory region or fill empty spaces.

In addition to the key path information and the type, the original JSON column is needed as the relation could contain multiple JSON columns. Moreover, JSON tiles stores the information on whether the key path is used with another type and whether null values are possible. The type information is necessary for correctness since the same key can have different values across the database. This is particularly interesting for JSON tiles, as null entries are often avoided due to a fine-grained JSON tiles size. For further optimizations, shown in Sections 4.6 and 4.8, the key paths that are not extracted are stored as well. Because the number of keys may be large, we store the key paths in a bloom filter [35].

4.5 Access Expression Evaluation

Information about the availability of an extracted column is only known during the table scan of each JSON tile. Because JSON tiles only materializes the frequent structures, not all keys are stored as columns. Therefore, the access on the raw JSON data must be performed if no extract is found.

Accesses on JSON tiles use the information stored in the header of each tile to find the correct position of the requested data. The key path is stored as a string with information on the nesting depth (the number of nested levels) and the size of the string. We compute the matching of key paths in linear time, as the number of different key paths is limited within a single tile. Since it is expensive to calculate the availability of materialized columns per tuple, the calculations is performed once per tile. It is cached and reused for all the following tuples of the same tile.

If a materialized column is available, the header of JSON tiles is used to compute the access information. The matching column start position is computed by the position of the tile data and the offset into the matching column. The type information of the column is used to load the data and determine the best cast options. In Section 4.3, we show the rewriting of the cast expression that is used by both JSON tiles and our binary representation (Section 5.4).

To find the correct materialized column, our algorithm uses the key path and the requested types as inputs. If the types do not match, we test whether the types are both numerical values or the request type is a cast to Text. The former type suggests that it is easy to cast between the extracted and desired value. The only exceptions are values of type Date or Time. These are not allowed to be transformed to a textual representation. This restriction is explained further in Section 4.9.

4.6 Optimizer Integration

The query optimizer relies on cardinalities and selectivities to find a suitable query plan. In particular, join ordering relies on statistics to minimize intermediate join results. Without any tile statistics, the content of the JSON column is completely opaque to the database. Consequently, the optimizer has no information on how often a key path exists in a document and on the possible values. This can result in poor query plans and slow queries – in particular for complex, multi-table queries.

When constructing JSON tiles, we gather additional information for each tile. However, for join ordering the information needs to be available for the complete table. Thus, the information of the individual tiles is aggregated to leverage the data insights during query optimization. The additional tile information is used for optimizations as discussed in Section 4.8.

In the following, we describe the steps necessary to provide per-column statistics and estimators for JSON tiles. We use a fixed number of frequency counters and HyperLogLog [25] sketches for the extracted paths. The frequency counters are used to argue about the cardinality of the keys in the data. If, for example, a query requests `replies is not null` from the tweet data of Figure 2, only 5 out of 8 tuples match. Our JSON tiles implementation collects HyperLogLog sketches as these are the primary source of domain statistics in Umbra. The collection of regular histograms would work analogously. We suggest 64 sketches and 256 frequency counters as an upper bound on the statistics to restrict the maximum amount of memory used for query optimization.

During frequent itemset mining (Section 3.3), the frequency of all key paths within a tile is computed. The frequency of the key paths is used as the starting point for itemset mining. Each entry of the database consists of the key and the number of occurrences in the tile. As described in Section 4.4, this database is also stored in the tile header. We update the relation-wide frequency counters (256 slots) if the key exists or replace empty slots as long as available. If all slots are utilized, we start replacing slots according to the most recent tile number that last updated that slot and the frequency count of the keys. Hence, new values can overwrite existing ones, however, the most frequent ones are always stored in the statistics.

To retrieve the cardinality of the keys, we simply use the frequency counters. If the key is not present in the frequency counters, we leverage the smallest available counter for this access. We argue that the missing counter will behave most similarly to the key with the minimal frequency of all retrieved counters. Although the smallest retrieved frequency is still an approximation, the results are significantly more accurate than using the global count of tuples.

Similar to the frequency counters, we collect statistics about the domain of the associated value of key paths with HyperLogLog sketches. When a tile is created, the inserted values are directly sampled. Hence, JSON tiles creates sketches without noticeable overhead. To aggregate the sketches at the relation level, we use the same replacement strategy as described with frequency counters. Note that HyperLogLog sketches are easy to combine.

During query optimization, the filter predicates on materialized JSON tiles leverage the distinct counts of the HyperLogLog sketches. With this information, better join orders are possible as the resulting join cardinality estimation is improved. Furthermore, different documents are sampled statically at query plan generation to find more accurate estimations. This improves the sketch estimates and creates new estimates if no HyperLogLog sketch is available.

4.7 Updates

As JSON tiles creates columnar chunks, we can simply update the values of the keys that were changed. As variable-length data is tracked in a separate memory region with offsets, value updates can be computed in place. If the new document does not contain some of the extracted keys, null values in the respective columns indicate the absence of these keys. Note that the tile header needs to add all new access paths to the bloom filter. Otherwise, queries that scan the data could incorrectly skip the changed tiles.

Tiles need to be recomputed only if many outlier documents are introduced. An outlier document is defined as one that does not overlap with the existing extracted keys. As the recomputation of the materialized JSON tiles are costly, the computation should only be triggered after the majority of the tuples do not match the current extracted JSON tiles schema.

4.8 Skip Tiles Without Matches

Because JSON tiles collects tuples locally, some tiles do not contain certain key paths. If an expression is searching for such a path, skipping these tiles seems valuable.

The simple skipping of tiles that do not provide a key path, similar to the previous work on efficient column stores [36, 53], leads to incorrect results. Accessing a value from a key that was not found returns a null value. Skipping null values results in incorrect results, for example, some aggregates count null values.

To overcome this issue, our system tracks the optimization path of skipping null values and whether null is evaluated as false. These two properties can change at an operator and expression level. For example, an inner-join on top of the access expression has the property that null values are skipped as the join condition is evaluated as false. Another example for skipping null values are comparisons, e.g., the expression `where x->>'key'::Float > 1`.

Thus, if the expression is not found and null values are skipped or evaluated as false, the whole JSON tile has no valuable information. These tiles are then skipped to improve performance.

4.9 Date and Time Extraction

Because the exact representation of values need to be restored during accesses, the extraction of Dates and Times from strings is complex. As many different formats exist, it is hard to guarantee the recreation of the original string. We use a hybrid method to store Dates and Times in which the access type is leveraged. If the database user casts the value into Date or Time, JSON tiles does not need to recreate the exact string representation. As a result, any correct internal representation can be used to satisfy the request. Therefore, JSON tiles extracts possible Date and Time values because these are probably accessed as such.

To find columns that store Dates and Times, we first sample on the potential column. If the string-encoded values match a Date or Time type, we extract these values encoded as SQL Timestamp. When the user casts the access to any Date- or Time-like type, the extracted Timestamp value is used to cast to the defined type. Otherwise, the string representation of the binary JSON is returned retaining the input format.

5 BINARY JSON FORMAT

JSON tiles extracts the frequent key paths of documents; however, some data sets contain outliers and infrequent keys that are not materialized. This section presents a new optimized binary format that allows fast access to individual keys of such infrequent objects. An optimized format is necessary as JSON is a human-readable data format. Each access results in an expensive parsing of the raw string. The goals for the binary format are fast lookups in objects and arrays, typed values, and few cache misses. The format must further conform to RFC 8259 [13], which defines the general JSON representation and needed value types.

Several binary JSON formats were developed to efficiently transfer data [26, 32, 49]. These formats, however, are not optimized for fast accesses and focus on (de-) serialization support. DBMS that use custom binary formats include PostgreSQL and MongoDB [45, 52]. Although the latter formats are better suited for query processing than exchange formats, they do not combine a logarithmic worst-case runtime for lookups with continuous memory accesses.

Our JSONB format is optimized to provide $O(log(n))$ accesses to the correct key in objects and $O(1)$ accesses to array elements. Moreover, objects and arrays are forward iterateable such that all key-value pairs – even nested objects – can be accessed continuously without memory address jumps. This results in fewer cache misses for nested accesses. The physical types used in our binary JSON representation match the RFC requirement and are also used by JSON tiles as mentioned in Section 3.3. Hence, the cast rewriting presented in Section 4.3 is a universal access optimization.

Due to restructuring, some of the unimportant properties of the original document get lost, such as whitespace information or the order of keys in the input. We argue, aligned with the guidelines of using binary JSON in PostgreSQL, that the gains in query performance outweigh the ability to recreate the exact syntax of the input [52]. Apart from the syntactic restriction, our JSONB document is round-trip safe. Thus, all other properties and the exact value representation can be reconstructed.

5.1 JSONB Storage Format

To provide interoperability and correctness, our binary JSON format conforms to RFC 8259 [13]. It defines objects, arrays, numbers, strings, and three literals. Our binary format uses the following data types to represent the type definitions. Each type has an 8-bit header with the type identifier and additional information.

Numeric Integers use the SQL type BigInt. We store small values (< 23) in the integer header, otherwise, we calculate the number of bytes needed to represent the integer and store the amount in the header. The size-optimal integer follows the header.

Numeric Floats store the remaining numeric values, using the SQL Float datatype represented with IEEE 754 double precision. RFC 8259 relies on double-precision floats for the remaining numerics as they are "generally available", "widely used", and "good interoperability can be achieved" [13]. We further optimize for smaller precision levels (half-floats, and single-precision floats) if the conversion from double-precision floats is lossless.

Literals use a special header representing the value.

Objects contain all key-value pairs including nested objects and arrays. Objects need an object header, followed by an integer representing the number of elements in that object. The integer uses the minimum number of bytes as defined for Numeric Integers. This is followed by an offset into the object for every element. Each offset points to the end of the corresponding element. The offset key follows the payload in every element slot. Note that nested objects and arrays are also stored in the payload such that we can iterate over the object without memory address jumps. Keys are further sorted to accelerate lookups and guarantee $O(log(n))$ accesses since we can use binary search to find the correct entry. The representation is illustrated in Figure 6.

Arrays are stored similar to objects but do not use keys.

Strings are stored with the possibility to use unicode characters. The exact representation matches the RFC 8259 definition.

5.2 Detection of Numerics in Strings

As RFC 8259 does not specify any precision for JSON numbers, strings are usually used to preserve the exact representation. For example, monetary values should not be represented as floating-point values. As a result, a decimal-valued price is usually stored as string. We auto-detect numeric values hidden in strings and extend our binary JSON format with an additional numeric string type.

During the transformation, we check whether the complete string, except the start and end quotes, can be represented as an SQL Numeric. We first test whether the input string is a valid numerical value (digits, point, etc.). If so, round-trip safety is guaranteed since the exact representation can always be reconstructed with the help of scale and precision. Because the original input must be a string, the start and end quotes are simply added to the Numeric output. The key motivation is that strings that are representable as Numeric will probably be used as numeric values. Query execution benefits by performing a smaller number of expensive casts from strings.

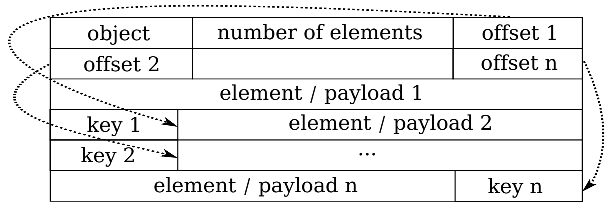


Figure 6: JSONB representation of an object.

5.3 Two-Pass Transformation Algorithm

Our JSONB format stores nested objects within the parent object. This allows for continuous accesses without memory address jumps. Continuously stored data increases locality and reduces cache misses. However, this makes object creation harder as the object size depends on the size of its nested objects and arrays. For example, the object with a nesting level of 0 only knows its size after the sizes of the inner objects have been computed. The simple approach of on-the-fly resizing is not feasible as resizing is expensive and needs to be performed for every inner object. Our compressed storage algorithm for floats and integers even aggravates this issue.

To overcome the problem of resizing, we propose a two-pass algorithm. In the first iteration, we check for validation errors and calculate the required memory for every JSONB type. This is possible because we remember the computed nesting level and perform a depth-first calculation. Note that depth-first is the order as defined in the input of JSON documents. Nested objects are textually represented within the parent object. Hence, we can simply forward iterate over the input. In the second iteration, we use the information of the first pass to allocate the right amount of memory and transform the data without further checks. In total, we iterate twice over the input data. However, this is usually not a performance issue because most JSON objects fit into the CPU cache.

5.4 Accessing Elements

Since JSON documents consist of objects and arrays that contain the information, the user typically looks up only specific parts of them. Umbra uses the access as JSONB -> and the access as Text ->> expressions to lookup the values of objects and arrays.

The access expression is implemented in two phases. First, a lookup into the object or array is executed. Object keys can be accessed in $O(log(n))$ since keys are sorted and binary search is used to perform the positioning. Because arrays are stored sequentially, we can access the element in $O(1)$. The second phase extracts the found value. The default extracted SQL types are JSONB (->) and Text (->>). As a result, the storage of the right type would reduce the performance if the access needs to cast to Text.

The database user usually casts the access result to the desired type, e.g., `x->>'key'::Integer`. Our system analyzes the cast and, if possible, directly returns the correct result type instead of the string representation. Otherwise, it parses the value as Text and performs the cast afterwards.

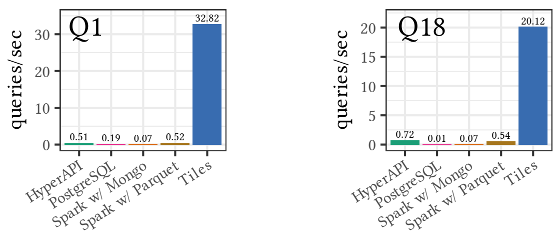


Figure 7: External competitors with all 32 threads.

Table 1: Execution times for all TPC-H queries in seconds.

	PG.	Spark		Hyper	Umbra			
		Mongo	Parquet		JSON	JSONB	Sinew	Tiles
1	5.276	14.297	1.939	1.950	1.725	0.178	0.122	**0.030**
2	> 100	23.383	2.735	1.370	1.608	0.584	0.637	**0.035**
3	17.905	15.892	1.288	0.560	0.675	0.280	0.259	**0.030**
4	3.013	10.439	1.755	0.539	0.692	0.227	0.228	**0.026**
5	87.468	22.659	2.072	> 100	1.340	0.372	0.326	**0.045**
6	1.259	14.896	0.690	0.244	0.254	0.119	0.085	**0.010**
7	> 100	21.035	2.554	3.111	1.177	0.429	0.351	**0.103**
8	> 100	26.608	1.814	1.156	1.469	0.474	0.416	**0.062**
9	> 100	23.688	3.939	1.728	2.576	0.395	0.370	**0.153**
10	> 100	21.967	2.003	0.984	1.362	0.388	0.294	**0.067**
11	> 100	23.444	0.809	0.829	1.070	0.344	0.353	**0.068**
12	1.493	18.783	1.316	0.419	0.450	0.286	0.289	**0.061**
13	5.570	10.597	2.146	0.683	0.665	0.149	0.291	**0.044**
14	1.502	9.552	0.734	0.343	0.392	0.171	0.142	**0.017**
15	9.105	19.024	1.306	0.339	0.399	0.211	0.185	**0.018**
16	4.220	15.119	2.693	0.898	0.629	0.201	0.273	**0.048**
17	> 100	16.379	1.381	0.605	0.567	0.173	0.091	**0.026**
18	86.167	14.861	1.849	1.388	0.949	0.260	0.179	**0.050**
19	1.290	33.885	0.970	0.363	1.834	0.213	0.170	**0.057**
20	> 100	20.234	1.613	0.787	0.974	0.355	0.348	**0.042**
21	12.372	39.236	3.517	1.415	1.787	0.615	0.479	**0.103**
22	2.060	11.306	3.135	0.529	0.566	0.172	0.180	**0.016**

6 EXPERIMENTAL EVALUATION

We integrated JSON tiles into our high-performance relational database system Umbra that supports SQL, columnar storage, and efficient memory management [23, 47]. We compare it with the following industrial-strength database systems: *PostgreSQL* (12.4) with its binary JSONB format, *Tableau Hyper* (0.11556) with its JSON format (no binary JSON available), *Apache Spark* (3.0) *with Apache Parquet* (Dremel), and *Apache Spark with MongoDB* (3.6). Because MongoDB does not support joins, Spark is used to schedule the queries. The mandatory sampling of the MongoDB data is not accounted.

Besides this system-wide comparison, we also integrated a number of prior JSON handling proposals into our system (sharing the optimizer and query engine): human-readable *JSON* format, our binary *JSONB* representation as described in Section 5, *Sinew*, which extracts the whole table with the original proposed 60% table-frequency using our JSONB format, and *JSON tiles* with JSONB. Unless otherwise noted, we use the tile size 2^{10}, partition size 8, and extraction threshold 60%.

All experiments were performed on an AMD Ryzen Threadripper 1950X (16 cores, 32 threads) with 64 GB of main memory. The system runs Ubuntu 20.04 and uses a Samsung 850 Pro SSD (2TB).

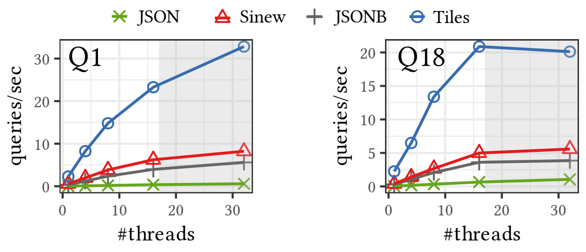


Figure 8: Scalability of internal competitors.

6.1 Combined TPC-H JSON

Our initial experiments are based on TPC-H. As this benchmark is based on a relational schema, we first explain the steps necessary to convert the data to JSON. Queries are modified similarly to the example query shown in Section 4.2. We modify TPC-H such that every row of each table is represented as a JSON object with the column names as the keys of the object. Thus, each JSON document contains the schema of the table and the values of one row. To simulate a combined log data workload with different JSON documents, we combine the different structures of these multiple relations into a single one. Although the documents are adapted in JSONized TPC-H, the queries return the same result as queries executed on the original TPC-H relations. Data loading is performed in parallel and uses all cores which leads to an imperfect insertion order.

In the following, we focus on chokepoints for TPC-H, which have been elaborated by previous work [11, 21]. Therefore, the results of the queries Q1 (expression calculation & aggregation), Q3 (join & aggregation), and Q18 (join) are shown in detail. The execution times of all TPC-H queries are shown in Table 1.

Query 1 only accesses items of the original lineitem table and performs low-cardinality aggregations with expensive expression calculations. As visually illustrated in Figure 7, our approach is an order of magnitude faster than Spark with Parquet and Hyper, which are both able to leverage a large fraction of the available cores. In comparison to other approaches within Umbra, visualized in Figure 8, we are able to speed up the computation by a factor of 3. As Query 1 relies on date expressions, our date and time optimization helps to significantly outperform Sinew. Umbra scales with a rising number of cores despite executing a single pipeline.

On the other hand, Query 18 joins multiple original tables with groups, and is therefore a chokepoint for join and high-cardinality aggregation performance. PostgreSQL uses a sub-optimal join order which results in very low performance. Although the lineitem columns accessed by the query are extracted with Sinew, the query performance is 4× slower than JSON tiles. This is a result of the missing information on cardinalities and the non-materialized tuples of customer and order data. Query 3 contains an expensive aggregation and performs joins. Our approach dominates all others since the optimal join order is computed and all lineitem fields are materialized.

6.2 Combined Yelp

To confirm the findings of the TPC-H benchmark, we test additional queries on the real-world Yelp data set (~9 GB) [64]. We define five queries on top of the data to gather interesting business insights [22]. Table 2 shows the results for all Yelp queries. For example, Yelp

Table 2: Execution times for all Yelp queries in seconds.

	PG.	Spark		Hyper	Umbra			
		Mongo	Parquet		JSON	JSONB	Sinew	Tiles
1	15.883	9.211	1.114	1.892	6.068	0.487	0.366	**0.293**
2	5.121	8.582	1.868	0.454	0.813	0.191	0.163	**0.044**
3	> 100	> 100	> 100	> 100	3.262	0.444	0.302	**0.145**
4	10.961	4.774	0.188	0.296	0.843	0.105	**0.013**	**0.013**
5	49.033	8.521	1.499	1.095	2.698	0.273	0.160	**0.088**

Table 3: Execution times for all Twitter queries in seconds.

	Spark		Hyper	Umbra				
	Mongo	Parquet		JSON	JSONB	Sinew	Tiles	Tiles-*
1	17.226	3.246	65.381	8.319	0.419	0.255	**0.116**	**0.116**
2	5.517	1.100	1.262	4.510	0.181	0.191	**0.091**	**0.091**
3	1.881	1.336	> 100	> 100	0.191	0.204	0.215	**0.017**
4	28.860	4.139	1.401	23.749	0.229	0.212	0.206	**0.022**
5	17.095	2.542	1.603	2.802	0.164	**0.049**	0.057	0.058

Query 4 counts the number of reviews in groups of stars. Because the number of reviews is large, Sinew also materializes all fields needed for this query. The performance of our approach and Sinew is very similar in this example, which results from the extraction of the star rating. Although this is one of the best cases for Sinew, our approach is able to slightly increase the performance, which highlights the small static overhead per JSON tile. JSON tiles has a higher throughput due to the skipping defined in Section 4.8.

6.3 Twitter

As we use Twitter as our running example, we benchmark multiple queries on an excerpt of tweets from June 1, 2020 (~31 GB) [22, 58]. Tiles-* combines JSON tiles with extracting high-cardinality arrays as discussed in Section 3.5. We extract high-cardinality arrays (hashtags, mentions) and store them in an additional JSON tiles relation. Queries join these relations with the original Twitter table.

Query 1 selects the tweets of the most influential users of the day. Although the user object is mandatory in tweets and extracted by both Tiles and Sinew, we are able to outperform the competitors. The deleted tweets of each user are aggregated with query 2. Deletions use a different JSON structure that is not frequent globally. This structure is reordered and can be materialized in some tiles.

Query 3 selects tweets that mention @ladygaga (user_mentions array), and query 4 selects tweets that include the hashtag #COVID (hashtags array). As both rely on the extraction of high-cardinality arrays, only a subset of the items is materialized within JSON tiles. JSON Tiles-* outperforms all competitors by joining the matching high-cardinality arrays with the base Twitter data.

Table 4: Geo-mean of Twitter.

	JSON	JSONB	Sinew	Tiles	Tiles-*
Twitter	11.803	0.258	0.239	0.122	0.054
Changing	11.683	0.236	0.182	0.115	0.054

Table 4 shows the geo-mean runtime on a data set that changes its tweet structure as described in Section 2.2 [22, 61]. As the changes in the JSON structure reduce the number of matches, most systems have an improved geometric mean. JSON tiles can easily adopt to unseen access keys and does not introduce null values if an access path is absent.

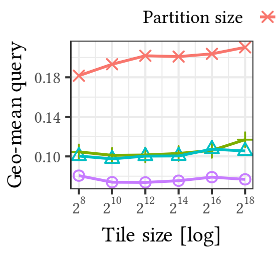


Figure 10: Geometric Mean of shuffled TPC-H.

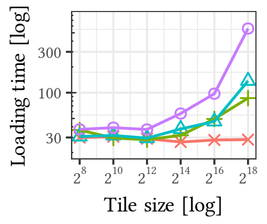


Figure 11: Loading Time of shuffled TPC-H.

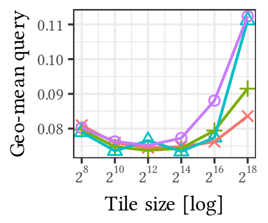


Figure 12: Yelp Geo-mean.

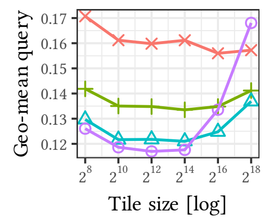


Figure 13: Tweet Geo-mean.

6.4 Shuffled TPC-H

To demonstrate the robustness of our novel partitioning algorithm, we manually shuffled the TPC-H table before loading. Thus, during the insertion no local tuple patterns are retained. JSON tiles with a partitioning of 8 and a tile size of 2^{10} is able to reduce the query runtime significantly. Figure 9 shows the geometric mean of the shuffled TPC-H benchmark. The JSON string representation has poor performance due to the parsing needed for every document. Although JSONB and Sinew are able to significantly increase performance, JSON tiles can further improve on these results by a factor of 4x.

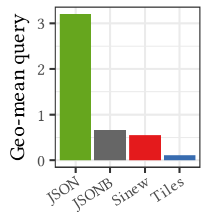


Figure 9: Shuffled TPC-H Geo-mean.

6.5 Tile and Partition Size

The choice of the tile and partition size has impact on the insertion time and materialization quality. The following experiments show how to find robust values for these settings.

Figures 10, 12, and 13 show the different choices and the resulting geometric mean for the respective workloads. The more partitions are enabled, the better the reordering. Even in naturally ordered data sets (e.g., Yelp and sequential TPC-H), the parallel insertion into our database (32 threads) creates outliers and imperfect data. Hence, the reordering is also beneficial there. Considering the insertion performance, Figure 11 highlights that a tile size of less than 2^{14} and a partition size of less than or equal to 8 do not introduce any overhead. Thus, we recommend tile size 2^{10} and partition size 8.

6.6 Optimizations for JSON Tiles

For the TPC-H and Yelp workloads, Figure 14 shows the impact of the optimizations discussed in Section 4.8 and 4.9. The skipping of tiles without matches is an optimization that helps to speed computations if the number of different JSON document types is higher.

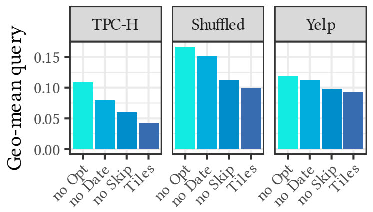


Figure 14: Geometric means of different optimization levels.

Because many real-world workloads use queries that are constrained by date ranges, the extraction of date and time is beneficial. As Figure 14 shows, the optimizations improve the performance considerably.

6.7 Micro Benchmark

The following micro benchmark demonstrates that our approach has only minimal static overhead for each JSON tile while gaining robustness. JSON tiles is able to achieve an order of magnitude improvements in comparison to only using binary JSON documents.

To explain the overhead behavior, we choose a query that is executed optimally by both the regular relational system and Sinew. The query simply sums up the linenumber field. Sinew extracts the column perfectly just like JSON tiles. The performance is shown in Figure 15, the corresponding low-level CPU performance counters are shown in Table 5. The benchmarks labeled with "Comb." use the combined TPC-H, whereas the others use the original lineitem table. Note that the relational approach cannot use combined TPC-H.

First, the materialization of JSON tiles leads to significant improvements over using the raw or binary optimized JSON formats. The performance of both extraction algorithms is similar to a pure relational TPC-H workload if the original lineitem table is used. The imperfect combined data consists of outliers and different structures because of the parallel data loading. The performance is reduced for the extraction algorithms, however, it is still an order of magnitude faster than when only JSONB is used. The relational table needs 32 instructions per tuple, Sinew 65, and JSON tiles 70. As this is the perfect extraction workload for Sinew, it is expected that the increased robustness requires some additional computations.

Table 5: Low-level performance counters for the summation query on lineitem; normalized per tuple computed.

System	Cycles	Instr.	Branch-	L1-Miss	Sec/All
Relational	17.01	31.58	0.00	0.02	0.001613
Tiles	39.33	69.82	0.02	0.18	0.002494
Sinew	32.12	65.08	0.01	0.10	0.002050
Sinew Comb.	39.07	71.73	0.03	0.10	0.003450
Tiles Comb.	50.15	74.20	0.04	0.14	0.004462

6.8 Data Loading and Storage Consumption

As our approach preprocesses the data during insertion, we measure the time needed to load the data sets. Figure 17 shows the loading times of all systems. The fastest system for TPC-H and Yelp is Hyper, which just stores the raw JSON string in the database and uses almost-instant data file loading [46].

Focusing on the overhead of JSON tiles, only a small reduction compared to the raw JSON and binary JSON insertion times are noticeable. The performance drop by Sinew results from the single-threaded frequency algorithm and the materialization of the detected columns. For a fair comparison, Sinew eagerly extracts the data after the insertion.

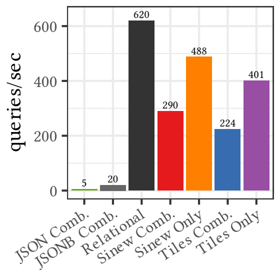


Figure 15: Throughput of the summation query.

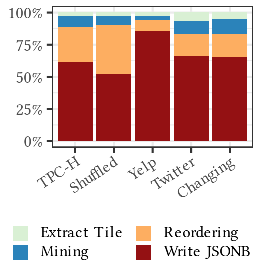


Figure 16: Insertion time breakdown (32 threads).

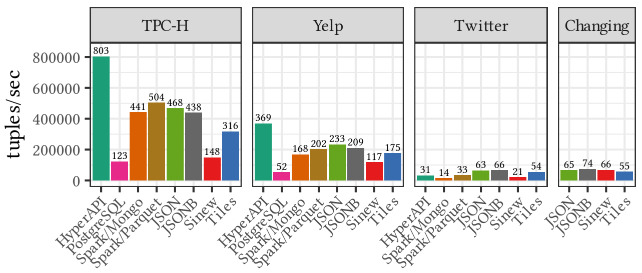


Figure 17: Parallel loading (numbers in 1000 tuples/sec).

Figure 16 breaks down the time needed for the different steps to create JSON tiles. Most of the insertion time is spent for storing the binary JSON data. Note that the expensive creation of the binary JSON is not measured as these steps are further up the pipeline. Although the JSON tiles operations require computation time, the overall loading times indicate that these computations do not change the insertion speed significantly. For example, the shuffled TPC-H spends a crucial amount of time on reordering, however, Figure 11 shows that this does not result in slower overall insertion times. Also, the insertion times do not change between partition sizes for small tile sizes.

Table 6: Size in MB (% of JSONB).

	JSON	JSONB	+Tiles	+LZ4-Tiles
TPC-H	3092	2766	665 (24%)	317 (11%)
Yelp	8657	7809	718 (9%)	237 (3%)
Twitter	31271	24106	706 (3%)	247 (1%)

We measured the size needed to store JSON tiles in Table 6 to analyze the storage requirements. In our current implementation, JSON tiles are materialized in addition to the original JSONB data. All benefits of JSON tiles come with only a moderate size overhead. As TPC-H consists of few strings and many extractable columns, the overhead is the highest there. Only 3% overhead results in significantly improved performance for Twitter. Because the data of JSON tiles are stored in columnar format, we can achieve strong compression ratios. For example, LZ4 compression on JSON tiles can further reduce storage consumption by a factor of 2-3x.

6.9 JSON Binary Formats with Nesting

As some documents cannot be extracted, we rely on a high-performance binary JSON representation. We compare our binary format, referred to as JSONB below, to the BSON implementation from MongoDB's open source C++ driver [45], and the JsonCons

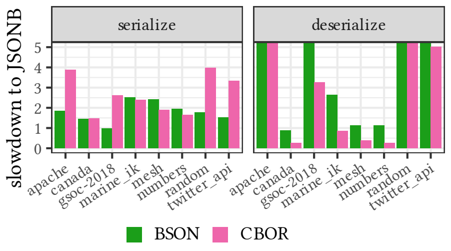


Figure 18: (De-) Serialization performance slowdown compared to our JSONB format.

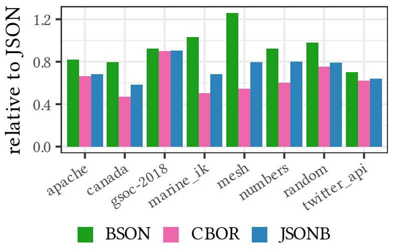


Figure 19: Storage size consumption compared to JSON string.

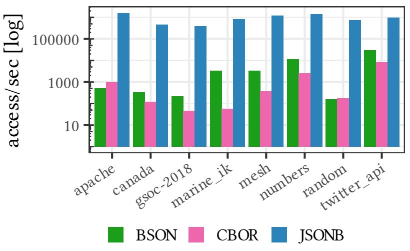


Figure 20: Performance of random accesses with different nesting levels.

C++ CBOR implementation [49]. To demonstrate a wide variety of complex and nested JSON documents, we use standardized JSON files from the SIMD-JSON repository [37].

First, we analyze the serialization and deserialization performance of the different JSON formats. Figure 18 shows the slowdown of the other two approaches compared to our JSONB implementation. JSONB is the fastest format in all serialization workloads and only three deserialization workloads are beneficial for CBOR.

The normalized costs for storing the JSON documents in a binary format are shown in Figure 19. CBOR's space requirements are the lowest as this is used mostly to exchange messages. In comparison to MongoDB's BSON, our representation uses less disk space.

Our binary representation has the best lookup performance, which is shown in Figure 20. Accessing keys within a document requires the object to be extracted in CBOR. This reduces the access performance significantly. Our $O(log(n))$ object key lookup is superior to the linear-time algorithm of BSON. Thus, JSONB achieves large performance gains for random accesses.

7 RELATED WORK

Due to the increasing importance of semi-structured data, many systems have been developed to handle different data documents. In the following, we differentiate between database systems and raw data processing systems.

Database Systems with JSON Support: With the rising usage of JSON, relational database systems integrate storage solutions for these data formats. One common idea is to store and index the data such that consequent accesses can be evaluated efficiently [3, 15, 18, 33, 40, 41, 56, 62]. Sinew [57] extends PostgreSQL with the approach of extracting data from the whole table, which incurs robustness problems for changing or combined data. This reduces query performance as only a certain number of keys are extracted [51, 57]. Our system focuses on the efficient and robust storage of JSON data to satisfy multiple user queries thereafter.

Proteus [33] builds indexes on top of JSON data to speed up accesses. Recache accelerates processing of heterogeneous formats by caching accesses of the data according to the query workload [8]. Other systems, such as Apache Spark [7] or Hive [59], use different storage plugins for heterogeneous data. Apache Parquet [6] and Avro [5] are common formats for storing JSON data. Although these plugins are quite robust, e.g., record shredding of Dremel [42], the performance of Spark on combined data is severely reduced.

NoSQL systems such as MongoDB [44], Couchbase [17], and DocumentDB [4] store semi-structured documents directly. However, their feature set for querying is limited and analytical (columnar) accesses are slow as these systems are optimized for point accesses.

Raw Data Processing: Another approach of accessing JSON files is to query raw files without explicitly loading them. After defining the queries and providing the raw files, the system should return the results without any loading delay [30]. Modern database systems try to saturate the wire speed to keep the loading gap small. Raw systems have reduced performance on multiple queries as the data is not stored as efficiently as possible [46].

Other approaches, e.g., NoDB [2], use in-situ raw accesses [10, 16, 50] to query the raw files directly. This requires the data to be parsed quickly. For both structured and semi-structured data, parsers such as FAD.js, Mison or SIMD-JSON use modern CPU properties for fast reads [12, 27, 37, 39]. Raw filters are used to speed up the parsing and reduce the amount of data ingested into the database [48, 63].

JSON Schema Retrieval: Inspired by the usage of *JSON Schema* [31], which is a work-in-progress description language for JSON, recent theoretical work [9, 20] has studied schema inference for JSON data. Although these approaches can describe the inherent JSON schema accurately, the computation of the schema file is expensive as all optional and required schema fields have to be enumerated. Different JSON documents in large-scale data sets can further decrease the performance, as the existence of many optional fields makes it harder to choose the right fields to extract.

8 CONCLUSION

We presented JSON tiles, a collection of algorithms and techniques for deeply integrating high-performance JSON support into relational database systems. High scan performance is achieved by extracting the frequent parts of the data into chunks of materialized JSON data. During the materialization we collect statistics about the data so that the query optimizer of the RDBMS is able to find good query plans. The materialized chunks are robust to heterogeneous data as we find globally and locally frequent structures. We further infer data types from the textual representation. If attributes cannot be extracted, we use an optimized binary format for JSON so that object lookups are in logarithmic time of the keys within an object. The experimental evaluation shows that our approach is an order of magnitude faster on imperfect and combined workloads, without adding any significant overhead to perfectly-structured data.

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *VLDB*. 487–499.

[2] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*. 241–252.

[3] Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey. 2020. An LSM-based Tuple Compaction Framework for Apache AsterixDB. *PVLDB* 13, 9 (2020), 1388–1400.

[4] Amazon Web Services. 2020. Amazon DocumentDB. https://aws.amazon.com/documentdb. accessed: 2020-01-03.

[5] Apache Software Foundation. 2020. Apache Avro. https://avro.apache.org/. accessed: 2020-01-04.

[6] Apache Software Foundation. 2020. Apache Parquet. https://parquet.apache.org. accessed: 2020-01-03.

[7] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *SIGMOD*. 1383–1394.

[8] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. 2017. ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data. *PVLDB* 11, 3 (2017), 324–337.

[9] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. 2017. Schema Inference for Massive JSON Datasets. In *EDBT*. 222–233.

[10] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. 2014. Parallel data analysis directly on scientific file formats. In *SIGMOD*. 385–396.

[11] Peter A. Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*. 61–76.

[12] Daniele Bonetta and Matthias Brantner. 2017. FAD.js: Fast JSON Data Access Using JIT-based Speculative Optimizations. *PVLDB* 10, 12 (2017), 1778–1789.

[13] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259.

[14] Luca Canali. 2017. Performance Analysis of a CPU-Intensive Workload in Apache Spark. https://externaltable.blogspot.com/2017/09/performance-analysis-of-cpu-intensive.html. Results presented at Spark Summit.

[15] Craig Chasseur, Yinan Li, and Jignesh M. Patel. 2013. Enabling JSON Document Stores in Relational Systems. In *WebDB*. 1–6.

[16] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *SIGMOD*. 1287–1298.

[17] Couchbase. 2019. Couchbase Under the Hood: An Architectural Overview. https://resources.couchbase.com/c/server-arc-overview.

[18] Benoît Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD*. 215–226.

[19] Alin Deutsch, Mary F. Fernández, and Dan Suciu. 1999. Storing Semistructured Data with STORED. In *SIGMOD*. 431–442.

[20] Michael DiScala and Daniel J. Abadi. 2016. Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data. In *SIGMOD*. 295–310.

[21] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *PVLDB* 13, 8 (2020), 1206–1220.

[22] Dominik Durner. 2019. JSON queries. https://github.com/durner/json-queries.

[23] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *DaMoN*. ACM, 21:1–21:3.

[24] Facebook. 2020. Using the Graph API. https://developers.facebook.com/docs/graph-api/using-graph-api. accessed: 2020-01-04.

[25] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. 137–156.

[26] Sadayuki Furuhashi. 2020. MessagePack. https://msgpack.org/. accessed: 2020-11-07.

[27] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative Distributed CSV Data Parsing for Big Data Analytics. In *SIGMOD*. 883–899.

[28] HackerNews. 2020. HackerNews Items API. https://github.com/HackerNews/API/. accessed: 2020-01-07.

[29] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining Frequent Patterns without Candidate Generation. In *SIGMOD*. 1–12.

[30] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. 2011. Here are my Data Files. Here are my Queries. Where are my Results?. In *CIDR*. 57–68.

[31] JSON Schema. 2020. Specification of the new draft. https://json-schema.org/specification.html. accessed: 2020-01-04.

[32] Riyad Kalla. 2020. UBJSON. https://ubjson.org/. accessed: 2020-11-07.

[33] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. 2016. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB* 9, 12 (2016), 972–983.

[34] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *The VLDB Journal* (2021).

[35] Adam Kirsch and Michael Mitzenmacher. 2008. Less Hashing, Same Performance: Building a Better Bloom Filter. *Random Structures & Algorithms* 33, 2 (2008), 187–218.

[36] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.

[37] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.

[38] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215.

[39] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *PVLDB* 10, 10 (2017), 1118–1129.

[40] Zhen Hua Liu and Dieter Gawlick. 2015. Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -. In *CIDR*.

[41] Zhen Hua Liu, Beda Christoph Hammerschmidt, and Doug McMahon. 2014. JSON data management: supporting schema-less development in RDBMS. In *SIGMOD*. 1247–1258.

[42] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010), 330–339.

[43] Tova Milo. 2019. Getting Rid of Data. https://vldb2019.github.io/files/VLDB19-keynote-2-slides.pdf. VLDB Keynote.

[44] MongoDB, Inc. 2019. MongoDB Architecture Guide. https://www.mongodb.com/collateral/mongodb-architecture-guide.

[45] MongoDB, Inc. 2020. Mongo CXX Driver. https://github.com/mongodb/mongo-cxx-driver/tree/r3.5.1.

[46] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant Loading for Main Memory Databases. *PVLDB* 6, 14 (2013), 1702–1713.

[47] Thomas Neumann and Michael Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[48] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *PVLDB* 11, 11 (2018), 1576–1589.

[49] Daniel Parker. 2019. JsonCons. https://github.com/danielaparker/jsoncons.

[50] Christina Pavlopoulou, E. Preston Carman Jr., Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. 2018. A Parallel and Scalable Processor for JSON Data. In *EDBT*. 576–587.

[51] PostgreSQL Global Development Group. 2020. JSON Functions and Operators. https://www.postgresql.org/docs/11/functions-json.html. accessed: 2020-01-03.

[52] PostgreSQL Global Development Group. 2020. JSON Types. https://www.postgresql.org/docs/11/datatype-json.html. accessed: 2020-01-03.

[53] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091.

[54] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. 1999. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*. 302–314.

[55] Splunk Inc. 2020. The Data-to-Everything Platform. https://www.splunk.com/. accessed: 2020-01-17.

[56] Tableau. 2020. Tableau Hyper API. https://help.tableau.com/current/api/hyper_api. accessed: 2020-01-04.

[57] Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. 2014. Sinew: a SQL system for multi-structured data. In *SIGMOD*. 815–826.

[58] Archive Team. 2020. The Twitter Stream Grab - 2020.06. https://archive.org/details/archiveteam-twitter-stream-2020-06. accessed: 2020-10-12.

[59] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009), 1626–1629.

[60] Twitter. 2020. Developer Guide. https://developer.twitter.com. accessed: 2020-01-03.

[61] Twitter. 2020. Tweet Timeline. https://developer.twitter.com/en/docs/tweets/data-dictionary/guides/tweet-timeline. accessed: 2020-01-07.

[62] Zhiyi Wang, Dongyan Zhou, and Shimin Chen. 2017. STEED: An Analytical Database System for TrEE-structured Data. *PVLDB* 10, 12 (2017), 1897–1900.
[63] Dong Xie, Badrish Chandramouli, Yinan Li, and Donald Kossmann. 2019. Fish-Store: Faster Ingestion with Subset Hashing. In *SIGMOD*. 1711–1728.
[64] Yelp. 2019. Yelp Dataset Challenge. https://www.yelp.com/dataset/challenge. accessed: 2019-11-20.

PAPER P4

Experimental Study of Memory Allocation for High-Performance Query Processing

Synopsis. Modern database engines can improve query processing by an order of magnitude compared to traditional database systems. Components that were not of concern in legacy systems are becoming performance bottlenecks, such as memory allocation. Because modern systems rely on temporary data structures, such as hash tables, a considerable number of short-living memory of varying sizes is allocated. The large number of memory operations affect performance and are visible in performance profiling. Since modern servers have hundreds of general-purpose cores, simultaneously handling multiple memory operations is crucial.

Memory allocators sit in between the operating system and the database system. They use different strategies for allocating and deallocating; for example, delayed return of memory to the operating system for higher subsequent accesses. Naturally, the choice of the memory allocator strategy influences query processing. The paper classifies allocators based on their performance overhead, the scalability on large systems, the memory fairness to other processes, and the overall memory usage.

This paper performs the first comprehensive analysis of the impact of memory allocation on high-performance query processing. It examines five different memory allocators within the modern database system *Umbra* on analytical workloads, such as TPC-DS and TPC-H. After discussing the observed allocation patterns of the workload, different experiments analyze the four allocator characteristics. Uniform and exponentially distributed workloads simulate realistic workloads on three different servers (up to 4 NUMA nodes). Somewhat surprisingly, the results show that the choice of memory allocator significantly impacts performance characteristics. The paper recommends using *jemalloc* as the default memory allocator because it is the most versatile allocator in terms of performance, scalability, memory fairness, and memory efficiency.

Contributions. Dominik Durner contributed substantially to the content of the paper, in particular concerning the development of the proposed ideas, the implementation of the system, the evaluation, and authoring substantial parts of the paper.

Reference. Dominik Durner, Viktor Leis, and Thomas Neumann. "Experimental Study of Memory Allocation for High-Performance Query Processing". In: *ADMS@VLDB*. 2019, pp. 1–9

Miscellaneous. The short version of this paper was previously published at DaMoN in 2019 [DLN19b].

Experimental Study of Memory Allocation for High-Performance Query Processing

Dominik Durner
Technische Universität München
dominik.durner@tum.de

Viktor Leis
Friedrich-Schiller-Universität Jena
viktor.leis@uni-jena.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Somewhat surprisingly, the behavior of analytical query engines is crucially affected by the dynamic memory allocator used. Memory allocators highly influence performance, scalability, memory efficiency and memory fairness to other processes. In this work, we provide the first comprehensive experimental study that analyzes and explains the impact of memory allocation for high-performance query engines. We test five state-of-the-art dynamic memory allocators and discuss their strengths and weaknesses within our DBMS. The right allocator can increase the performance of TPC-DS (SF 100) by 2.7x on a 4-socket Intel Xeon server.

1. INTRODUCTION

Modern high-performance query engines are orders of magnitude faster than traditional database systems. As a result, components that hitherto were not crucial for performance may become a performance bottleneck. One such component is memory allocation. Most modern query engines are highly parallel and heavily rely on temporary hash-tables for query processing which results in a large number of short living memory allocations of varying size. Memory allocators therefore need to be scalable and be able to handle myriads of small and medium sized allocations as well as several huge allocations simultaneously. As we show in this paper, memory allocation has become a large factor in overall query processing performance.

New hardware trends exacerbate the allocation issues. The development of multi- and many-core server architectures with up to hundred general purpose cores is a distinct challenge for memory allocation strategies. Due to the increased number of pure computation power, more active queries are possible. Furthermore, multi-threaded data structure implementations lead to dense and simultaneous access patterns. Because most multi-node machines rely on a non-uniform memory access (NUMA) model, requesting memory from a remote node is particularly expensive.

10th International Workshop on Accelerating Analytics and Data Management Systems (ADMS'19), August 26, 2019, Los Angeles, California, CA, USA.

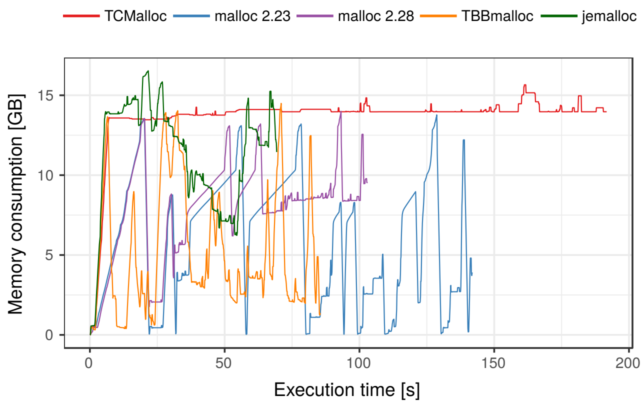


Figure 1: Execution of a given query set on TPC-DS (SF 100) with different allocators.

Therefore, the following goals should be accomplished by a dynamic memory allocator:

Performance: Minimize the overhead for malloc and free.
Scalability: Reduce overhead for multi-threaded allocs.
Memory Fairness: Give freed memory back to the OS.
Memory Efficiency: Reduce memory consumption.

In this paper, we perform the first comprehensive analysis that highlights and explains the impact of memory allocation in modern database systems. We evaluate different approaches to the aforementioned dynamic memory allocator requirements. Although memory allocation is on the critical path of query processing, no empirical study on different dynamic memory allocators for in-memory database systems has been conducted [1].

Figure 1 shows the effects of different allocation strategies on TPC-DS with scale factor 100. We measure memory consumption and execution time with our multi-threaded database system on a 4-socket Intel Xeon server. In this experiment, our DBMS executes the query set sequentially using all available cores. Even this relatively simple workload already results in significant performance and memory usage differences. Our database linked with `jemalloc` can reduce the execution time to $\frac{1}{2}$ in comparison to linking it with the standard `malloc` of glibc 2.23. Moreover, the used average and peak memory consumption of the allocators vary highly. Although the resident memory consumption seems high for `TCMalloc`, it already gives back the memory to the operating system lazily. Consequently, the allocation strategy is crucial to the performance and memory consumption behavior of in-memory database systems.

The remainder of this paper is structured as follows: After discussing related work in Section 2, we describe the used allocators and their most important design details in Section 3. Section 4 highlights important properties of our research DBMS “umbra” and analyzes the executed workload according to its allocation pattern. Our comprehensive experimental study is evaluated in Section 5. Section 6 summarizes our findings.

2. RELATED WORK

Ferreira et al. [9] analyzed dynamic memory allocators for a variety of multi-threaded workloads. However, the study considers only up to 4 cores. Therefore, it is hard to predict the scalability for today’s many-core systems.

In-memory DBMS and analytical query engines, such as HyPer [15], SAP HANA [20], and Quickstep [23] are built to utilize as many cores as possible to speed up query processing. Because these system rely on allocation-heavy operators (e.g., hash joins, aggregations), a revised experimental analysis on the scalability of the state-of-the-art allocators is needed. In-memory hash joins and aggregations can be implemented in many different ways which can influence the allocation pattern heavily [2, 3, 26, 17].

Some online transaction processing (OLTP) systems try to reduce the allocation overhead by managing their allocated memory in chunks to increase performance for small transactional queries [25, 24, 5]. However, many database systems process both transactional and analytical queries. Therefore, the wide variety of memory allocation patterns for analytical queries needs to be considered as well. Custom chunk memory managers help to reduce memory calls for small allocations but larger chunk sizes trade memory efficiency in favor of performance. Thus, our database system uses transaction-local chunks to speed up small allocations. Despite these optimizations, allocations are still a performance issue. Hence, the allocator choice is crucial to maximize throughput.

Preliminary results showed that memory allocation indeed has impact on the performance of query engines [4]. In this study, we analyze and explain the effects of different allocation strategies in order to understand all strengths and weaknesses of current allocators on modern hardware.

With the development of non-volatile memory (NVM), new allocation requirements were introduced. Foremost, the defragmentation and safe release of unused memory is important since all changes are persistent. New dynamic memory allocators for these novel persistent memory systems have been developed and experimentally studied [22]. However, regular allocators outperform these NVM allocators in most workloads due to fewer memory constraints.

3. MEMORY ALLOCATORS

In this section, we discuss the five different allocation strategies used for our experimental study. We explain the basic properties of these algorithms according to memory allocation and freeing. The tested state-of-the-art allocators are available as Ubuntu 18.10 packages. Only the glibc `malloc 2.23` implementation is part of a previous Ubuntu package. Nevertheless, this version is still used in many current distributions such as the stable Debian release.

Memory allocation is strongly connected with the operating system (OS). The mapping between physical and virtual memory is handled by the kernel. Allocators need to request virtual memory from the OS. Traditionally, the user program asks for memory by calling the malloc method of the allocator. The allocator either has memory available that is unused and suitable or needs to request new memory from the OS. For example, the Linux kernel has multiple APIs for requesting and freeing memory. `brk` calls can increase and decrease the amount of memory allocated to the data segment by changing the program break. `mmap` maps files into memory and implements demand paging such that physical pages are only allocated if used. With anonymous mappings, virtual memory that is not backed by a real file can be allocated within main memory as well. The memory allocation process is visualized below.

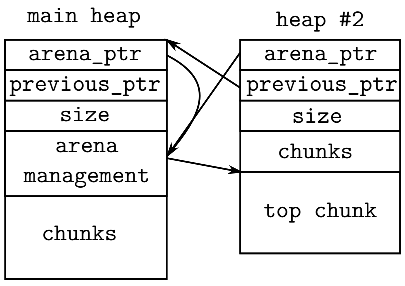
$$\begin{tikzcd}\boxed{\texttt{DBMS}} \arrow[r, "\texttt{malloc}", "\texttt{free}"'] & \boxed{\texttt{Allocator}} \arrow[r, "\texttt{mmap/brk}", "\texttt{munmap/brk}"'] & \boxed{\texttt{OS}}\end{tikzcd}$$

Besides freeing memory directly with the aforementioned calls, the memory allocator can opt to release memory with `MADV_FREE` (since Linux Kernel 4.5). `MADV_FREE` indicates that the kernel is allowed to reuse this memory region. However, the allocator can still access the virtual memory address and either receives the previous physical pages or the kernel provides new zeroed pages. Only if the kernel reassigns the physical pages, new ones need to be zeroed. Hence, `MADV_FREE` reduces the number of pages that require zeroing compared to regular freeing since the old pages might be reused by the same process.

3.1 malloc 2.23

The standard glibc `malloc` implementation is derived from `ptmalloc2` which originated from `dlmalloc` [19]. It uses chunks of various sizes that exist within a larger memory region known as the heap. `malloc` uses multiple heaps that grow within their address space.

For handling multi-threaded applications, `malloc` uses arenas that consist of multiple heaps to speed up simultaneous accesses. At program start the main arena is created and additional arenas are chained with previous arena pointers. The arena management is stored within the main heap of that arena. Additional arenas are created with `mmap` and are limited to eight times the number of CPU cores. For every allocation, an arena-wide mutex needs to be acquired. Within arenas free chunks are tracked with free-lists. Only if the top chunk (adjacent unmapped memory) is large enough, memory will be returned to the OS.



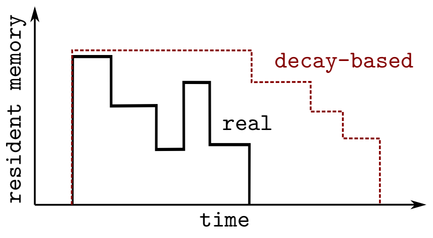
`malloc` is aware of multiple threads but no further multi-threaded optimizations, such as thread locality or NUMA awareness, is integrated. It assumes that the kernel handles these issues.

3.2 malloc 2.28

A thread-local cache (tcache) was introduced with glibc v2.26 [18]. This cache requires no locks and is therefore a fast path to allocate and free memory. If there is a suitable chunk in the tcache for allocation, it is directly returned to the caller bypassing the rest of the malloc routine. The deletion of a chunk works similarly. If the tcache has a free slot, the chunk is stored within it instead of immediately freeing it.

3.3 jemalloc 5.1

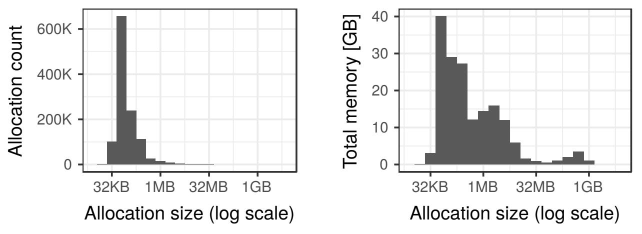
`jemalloc` was originally developed as scalable and low fragmentation standard allocator for FreeBSD. Today, it is used as the default allocator for a variety of applications such as Facebook, Cassandra and Android. It differentiates between three size categories - small (< 16KB), large (< 4MB) and huge. These categories are further split into different size classes. It uses arenas that act as completely independent allocators. Arenas consist of chunks that allocate multiples of 1024 pages (4MB). `jemalloc` implements low address reusage for large allocations to reduce fragmentation. Low address reusage, which basically scans for the first large enough free memory region, has similar theoretical properties as more expensive strategies such as best-fit. `jemalloc` tries to reduce zeroing of pages by deallocating pages with `MADV_FREE` instead of unmapping them. Most importantly, `jemalloc` purges dirty pages decay-based with a wall-clock (since v4.1) which leads to a high reusage of recently used dirty pages. The decay-based reclaiming frees pages that were not accessed for a certain time which is illustrated in the figure below. Consequently, the unused memory will be purged if not requested anymore to achieve memory fairness [6, 7].



3.4 TBBmalloc 2017 U7

Intel's Threading Building Blocks (TBB) allocator is based on the scalable memory allocator `McRT` [12]. It differentiates between small, quite large, and huge objects. Huge objects ($\geq$ 4MB) are directly allocated and freed from the OS. Small and large objects are organized in thread-local heaps with chunks stored in memory blocks.

Memory blocks are memory mapped regions that are multiples of the requested object size class and inserted into the global heap of free blocks. Freed memory blocks are stored within a global heap of abandoned blocks. If a thread-local heap needs additional memory blocks, it requests the memory from one of the global heaps. Memory regions are unmapped during coalescing of freed memory allocations if no block of the region is used anymore [16, 14].



(a) By number of allocations. (b) By memory consumption.

Figure 2: Allocations in TPC-DS (SF 100, serial execution).

3.5 TCMalloc 2.5

`TCMalloc` is part of Google's gperftools. Each thread has a local cache that is used to satisfy small ($\leq$ 256KB) allocations. Large objects are allocated in a central heap using 8KB pages.

`TCMalloc` uses different allocatable size classes for the small objects and stores the thread cache as a singly linked list for each of the size classes. Medium sized allocations ($\leq$ 1MB) use multiple pages and are handled by the central heap. If no space is available, the medium sized allocation is treated as a large allocation. For large allocations, spans of free memory pages are tracked within a red-black tree. A new allocation just searches the tree for the smallest fitting span. If no span is found, the memory is allocated from the kernel [10].

Unused memory is freed with the help of `MADV_FREE` calls. Small allocations are garbage collected if the thread-local cache exceeds a maximum size. Freed spans are immediately released since the "aggressive decommit" option was enabled (starting with version 2.3) to reduce memory fragmentation [11].

4. DBMS AND WORKLOAD ANALYSIS

Decision support systems rely on analytical queries that gather information from a huge dataset by joining different relations for example. In in-memory query engines joins are often scheduled physically as hash joins resulting in a large number of smaller allocations. In the following, we use a database system that uses pre-aggregation hash tables to perform multi-threaded group bys and joins [17]. Our DBMS has a custom transaction-local chunk allocator to speed up small allocations of less than 32KB. We store small allocations in chunks of medium sized memory blocks. Since only small allocations are stored within chunks, the memory efficiency footprint of these small object chunks is marginal. Additionally, the memory needed for tuple materialization is acquired in chunks. These chunks grow as more tuples are materialized. Thus, we already reduce the stress on the allocator significantly while preserving memory efficiency.

The TPC-H and TPC-DS benchmarks were developed to standardize common decision support workloads [21]. Because TPC-DS contains a larger workload of more complex queries than TPC-H, we focus on TPC-DS in the following. As a result, we expect to see a more diverse and challenging allocation pattern. TPC-DS describes a retail product supplier with different sales channels such as stores and web sales.

In the following, we statistically analyze the allocation pattern for TPC-DS executing all queries without rollup and

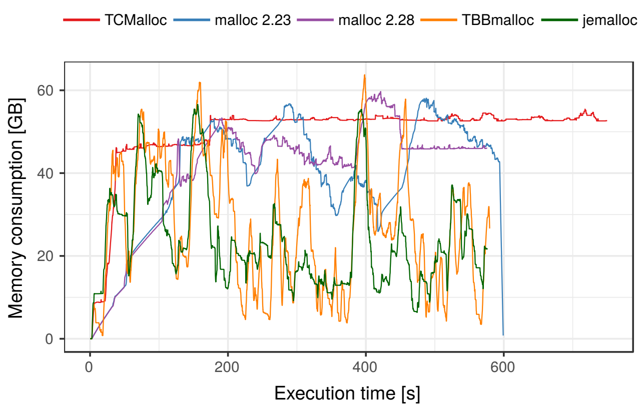


Figure 3: Memory consumption over time (4-socket Xeon, λ=1.25 q/s, SF 100).

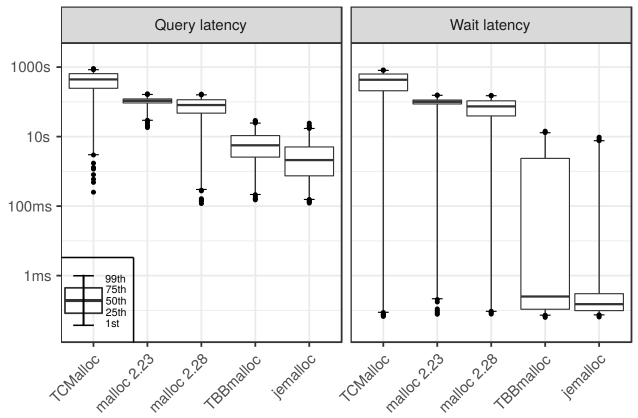


Figure 4: Total query latency and wait time (4-socket Xeon, λ=1.25 q/s, SF 100).

window functions. Note that the specific allocation pattern depends on the discussed implementation choices of the join and group by operators.

Figure 2 shows the distribution of allocations in our system for TPC-DS with scale factor 100. The most frequent allocations are in the range of 32KB to 512KB. Larger memory regions are needed to create the bucket arrays of the chaining hash tables. The huge amount of medium sized allocations are requested to materialize tuples using the aforementioned chunks.

Additionally, we measure which operators require the most allocations. The two main consumer are group by and join operators. The percentage of allocations per operator for a sequential execution of queries on TPC-DS (SF 100) is shown in the table below:

	Group By	Join	Set	Temp	Other
By Size	61.2%	25.7%	4.3%	8.4%	0.4%
By Count	77.9%	11.7%	8.5%	1.8%	0.1%

To simulate a realistic workload, we use an exponentially distributed workload to determine query arrival times. We sample from the exponential distribution to calculate the time between two events. An independent constant average rate λ defines the waiting time of the distribution. In comparison to a uniformly distributed allocation pattern, the number of concurrently active transactions varies. Thus, a more diverse and complex allocation pattern is created. The events happen within an expected time interval value of $1/\lambda$ and variance of $1/\lambda^2$. The executed queries of TPC-DS are uniformly distributed among the start events. Due to the usage of the same query-set and query arrival rates, we are able to test all allocators on the same real-world alike workloads.

Our main-memory query engine allows up to 10 transactions to be active simultaneously. If more than 10 transactions are queried, the transaction is delayed by the scheduler of our DBMS until the active transaction count is decreased. Due to intra-query parallelization, all cores of the system are utilized even with a reduced number of concurrent transactions.

5. EVALUATION

In this section, we evaluate the five allocators on three hardware architectures with different workloads. We show that the approaches have significant performance and scalability differences. Additionally, we compare the allocator implementations according to their memory consumption and release strategies which shows memory efficiency and memory fairness to other processes.

We test the allocators on a 4-socket Intel Xeon E7-4870 server (60 cores) with 1 TB of main memory, an AMD Threadripper 1950X (16 cores) with 64 GB main memory (32 GB connected to each die region), and a single-die Intel Core i9-7900X (10 cores) server with 128 GB main memory. All three systems support 2-way hyperthreading. These three different architectures are used to analyze the behavior in terms of the allocators' ability to scale on complex multi-socket NUMA systems. To avoid effects of loading the database into main memory, we use the second run of each workload to generate the execution graphs.

This section begins with a detailed analysis of a realistic workload on the 4-socket server. We continue our evaluation by scheduling a reduced and increased number of transactions to test the allocators' performance in varying stress scenarios. An experimental analysis on the different architectures gives insights on the scalability of the five malloc implementations. An evaluation of the memory consumption and the memory fairness to other processes concludes this section.

5.1 Memory Consumption and Query Latency

The first experiment measures an exponentially distributed workload to simulate a realistic query arrival pattern on the 4-socket Intel Xeon server. Figure 3 shows the memory consumption over time for TPC-DS (SF 100) and a constant query arrival rate of $\lambda = 1.25$ q/s. Although the same workload is executed, very different memory consumption patterns are measured. `TBBmalloc` and `jemalloc` release most of their memory after query execution. Both `malloc` implementations hold a minimum level of memory which increases over time. `TCMalloc` releases its memory accurately with `MADV_FREE` which is not visible by tracking the system provided resident memory of the database process. Due to huge performance degradations for tracking the lazy free-

Allocator	Local	Remote	Total	Page Fault
malloc 2.28	63B	172B	236B	41M
	100%	100%	100%	100%
jemalloc	120%	97%	103%	400%
TBBmalloc	121%	97%	103%	516%
TCMalloc	106%	105%	104%	153%
malloc 2.23	103%	100%	101%	139%

Table 1: NUMA-local and NUMA-remote DRAM accesses and OS page faults (4-socket Xeon, λ=1.25 q/s, SF 100).

ing of memory, we show the described release behavior of `TCMalloc` in Section 5.4 separately. However, the overall performance is reduced due to an increased number of kernel calls.

For an in-depth performance analysis, the query and wait latencies of the individual queries are visualized in Figure 4. Although the overall runtime is similar between different allocators, the individual query statistics show that only `jemalloc` has minor wait latencies. `TBBmalloc` and `jemalloc` are mostly bound by the actual execution of the query. On the contrary, both glibc `malloc` implementations and `TCMalloc` are dominated by the wait latencies. Thus, our database linked with the later allocators cannot process the queries fast enough to prevent query congestion. Query congestion results from the bound number (10) of concurrently scheduled transactions that our scheduler allows to be executed simultaneously.

Because of these huge performance differences, we measure NUMA relevant properties to highlight advantages and disadvantages of the algorithms. Table 1 shows page faults, local and remote DRAM accesses. All measurements are normalized to the current standard glibc `malloc 2.28` implementation for an easier comparison. The two fastest allocators have more local DRAM accesses and significantly more page faults, but have a reduced number of remote accesses. Note that the system requires more remote DRAM accesses due to NUMA-interleaved memory allocations of the TPC-DS base relations. Thus, the highly increased number of local accesses change the overall number of accesses only slightly. Minor page faults are not crucially critical since both `jemalloc` and `TBBmalloc` release and acquire their pages frequently. These minor page faults can be handled without disk I/O such as a request for a zeroed page [8]. Consequently, remote accesses for query processing are the major performance indicator. Because `TCMalloc` reuses `MADV_FREE` pages and therefore avoids unnecessary zeroing of pages, the number of minor page faults remains small.

5.2 Performance with Varying Stress Levels

In the previous workload, only two allocators were able to efficiently handle the incoming queries. This section evaluates the effects for a varying constant rate λ. We analyze two additional workloads that use the rates $\lambda = 0.63$ and $\lambda = 2.5$ queries per second. Thus, we respectively increase and decrease the average waiting time before a new query is scheduled by a factor of 2.

Figure 5 shows the query latencies of the three workloads. The results for the reduced and increased waiting times confirm the previous observations. The allocators

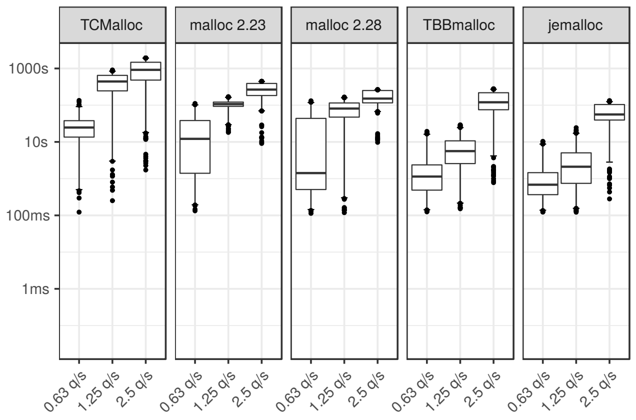


Figure 5: Query latency distributions for different query rates (4-socket Xeon, SF 100).

have the same respective latency order in all three experiments. `jemalloc` performs best again for all workloads, followed by `TBBmalloc`.

All query latencies are dominated by the wait latencies in the $\lambda = 2.5$ workload due to frequent congestions. With an increased waiting time ($\lambda = 0.63$) between queries, the glibc `malloc 2.28` implementation is able to reduce the median latency to a similar level as `TBBmalloc`. However, the query latencies within the third quantile vary vastly. `TCMalloc` and `malloc 2.23` are still not able to process the queries without introducing long waiting periods.

5.3 Scalability

After analyzing the allocators' perfromance on the 4-socket Intel Xeon architecture, this section focuses on the scalability of the five dynamic memory allocators. We execute an exponentially distributed workload with TPC-DS (SF 10) on the NUMA-scale 60 core Intel Xeon server, the 16 core AMD Threadripper (two die regions), and the single-socket 10 core Intel Skylake X.

Figure 6 shows the memory consumption during the workload execution. Since the AMD Threadripper has a very similar memory consumption pattern to the Intel Skylake X, we only show the 4-socket Intel Xeon and the single-socket Intel Skylake. Most notable are the differences of both glibc `malloc` implementations. These two allocators have a very long initialization phase on the 4-socket system, but are able to allocate their initial memory as fast as the other ones on the single-socket system. Due to more cores and the resulting different access pattern, the decay-based deallocation pattern of `jemalloc` differs slightly in the beginning. However, `jemalloc`'s decay-based purging reduces the memory consumption on both architectures considerably. `TCMalloc` cannot process all queries in the same time frame as the other allocators on the 4-socket system whereas it finishes at the same time on Skylake.

Especially the query latencies differ vastly between the architectures. In Figure 7, we show the latencies for the $\lambda = 6$ q/s workload. The more cores are utilized, the larger are the latency differences between the allocators. On the single-socket Skylake X, all the allocators have very similar performance. Besides having more cores, AMD's Threadripper uses two memory regions which requires a more ad-

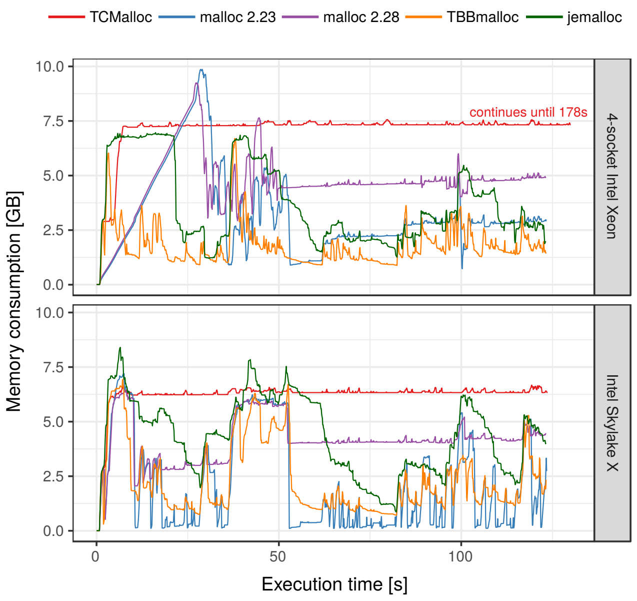


Figure 6: Memory consumption over time (λ=6 q/s, SF 10).

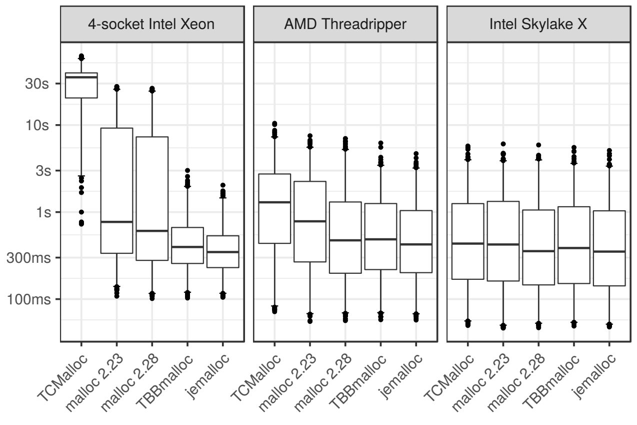


Figure 7: Query latencies (λ=6 q/s, SF 10).

vanced placement strategy to obtain fast accesses. In particular, `TCMalloc` and `malloc 2.23` without a thread-local cache have a reduced performance. The latency variances are reduced on the Threadripper but the overall latencies are worse in comparison to the Skylake architecture.

Yet, the most interesting behavior is introduced by the multi-socket Intel Xeon. It has both the best and worst overall query performance. `jemalloc` and `TBBmalloc` execute the queries with the overall lowest latencies and smallest variance. On the other hand, `TCMalloc` is worse by more than 10x in comparison to any other allocator. Both glibc implementations have a similar median performance but incur high variance such that a reliable query time prediction is impossible.

To substantiate our findings that remote accesses and large amount of cores are the major drivers, we evaluate the queries on a single-socket of the Intel Xeon server. We use `numactl` to bind the memory to the same region as the 30

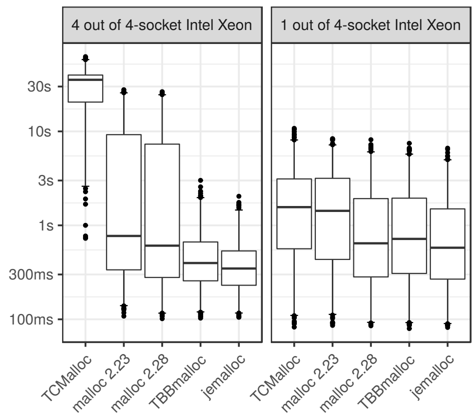


Figure 8: Query latencies (λ=6 q/s, SF 10).

Allocator	peak total request	measured[1]	average total request	measured
TCMalloc	55.7 GB	58.1 GB	17.8 GB	53.7 GB
malloc 2.23	61.4 GB	61.0 GB	26.2 GB	41.3 GB
malloc 2.28	61.5 GB	62.6 GB	20.2 GB	42.5 GB
TBBmalloc	55.7 GB	55.7 GB	15.9 GB	27.9 GB
jemalloc	58.6 GB	59.4 GB	11.1 GB	24.7 GB

Table 2: Memory usage (4-socket Xeon, λ=1.25 q/s, SF100).

threads used for execution. Figure 8 shows that the single-socket execution on our large system behaves similarly to the single-socket Skylake X.

The experiments show that both `jemalloc` and `TBBmalloc` are able to scale to large systems with many cores. `TCMalloc`, on the other hand, has significant performance loss on larger servers.

To validate our findings, we evaluate a subset of the queries on MonetDB 11.31.13 [13]. We observe a performance boost by using `jemalloc` on MonetDB; however, the differences are smaller because our DBMS parallelizes better and thus utilizes more cores.

5.4 Memory Fairness

Because DBMS often run alongside other processes on a single server, it is necessary that the query engines are fair to other processes. In particular, the memory consumption and the memory release pattern are good indicators of the allocators' memory fairness.

Our DBMS is able to track the allocated memory regions with almost no overhead. Hence, we can compare the measured process memory consumption with the requested one. The used memory differs between the allocators due to the performance and scalability properties although we execute the same set of queries. Table 2 shows the peak and average memory consumption for the $\lambda = 1.25$ q/s workload (SF 100) on the 4-socket Intel Xeon. We use the requested peak and average total memory consumption as the memory efficiency indicator of the allocators. The peak memory

[1]Due to chunk-wise allocation with unfaulted pages and measurement delays the measured amount of memory can be slightly smaller than the requested one.

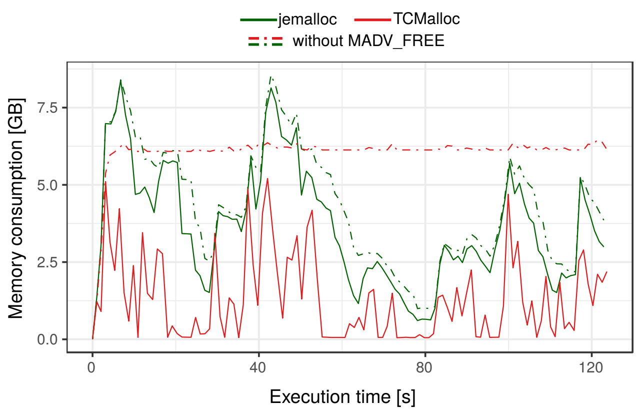


Figure 9: Memory consumption over time with subtracted `MADV_FREE` pages (λ=6 q/s, SF 10).

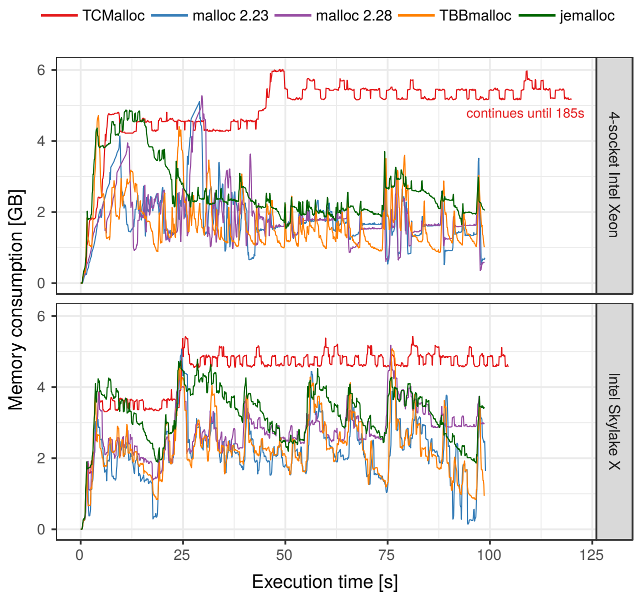


Figure 10: Memory consumption (λ=12 q/s, TPC-H SF 10).

consumption is similar for all tested allocators. On the contrary, the average consumption is highly dependant on the used allocator. Both glibc `malloc` implementations demand a large amount of average memory. `jemalloc` requires less average memory than `TBBmalloc`. However, the DBMS requested average memory is also higher for the allocators with increased memory usage. The higher average consumption results from an overall shorter execution time. Although the consumption of `TCMalloc` seems to be higher, it actually uses less memory than the other allocators. This results from the direct memory release with `MADV_FREE`. The tracking of `MADV_FREE` calls on the 4-socket Intel Xeon is very expensive and would introduce many anomalies for both performance and memory consumption. Therefore, we analyze the madvise behavior on the single-socket Skylake X that is only affected slightly by the `MADV_FREE` tracking. The memory consumption with the $\lambda = 6$ q/s workload (SF 10) is shown

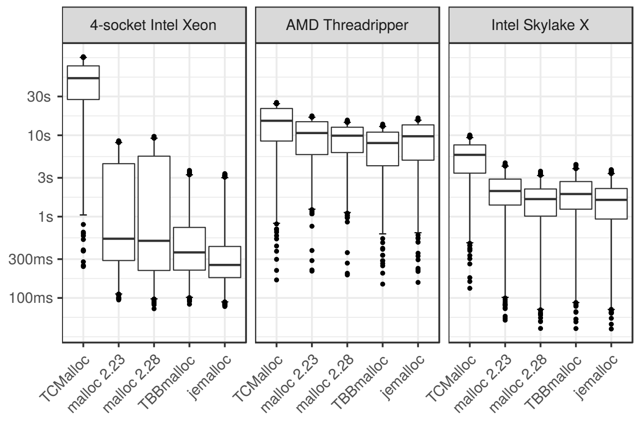


Figure 11: Query latencies (λ=12 q/s, TPC-H SF 10).

in Figure 9. The only two allocators that use `MADV_FREE` to release memory are `jemalloc` and `TCMalloc`. The measured average memory curve of `TCMalloc` follows the DBMS required curve almost perfectly. `jemalloc` has a 15% reduced consumption if the `MADV_FREE` pages are subtracted from the used memory.

5.5 Scalability with TPC-H

To validate our scalability results on analytical queries, we further analyze the TPC-H benchmark. We execute an exponentially distributed TPC-H (SF 10) workload on the NUMA-scale Intel Xeon server, the AMD Threadripper, and the Intel Skylake X.

Figure 10 shows the memory consumption over time executing the TPC-H workload. Due to an increased query rate, the allocation pattern is less smooth with TPC-H than with TPC-DS. `jemalloc` purges pages according to its decay strategy and the `malloc` implementations need an initial start-up phase. Thus, the allocation pattern of TPC-H is similar to the pattern of TPC-DS.

The query latencies for TPC-H are shown in Figure 11. Similar to our previous findings, `jemalloc` and `TBBmalloc` scale best. The allocators show only on the large Intel Xeon system huge performance differences.

5.6 Raw Allocation

Because our DBMS uses a custom chunkwise allocator with free-lists to speed up small allocations, we also evaluate the experiments without this 2-layered allocation setup. Every allocation request gets directly forwarded to the allocator instead of using the DBMS small allocation logic.

Figure 12 shows the query latencies of the three workloads for TPC-DS (SF 100) with direct allocator usage. `jemalloc` outperforms the other allocators. In comparison to the 2-layered allocation process, `TBBmalloc` cannot efficiently process the medium sized workload anymore. The other allocators behave similar, however the query latencies are increased. Overall, the usage of an additional small allocation is beneficial to reduce query processing time.

The memory consumption over time on the Intel Xeon and the Intel Skylake is shown in Figure 13. Interestingly, the release strategy of `TBBmalloc` is very different to the experiments with our additional small allocation logic. Regardless of the used system, the memory is only returned to

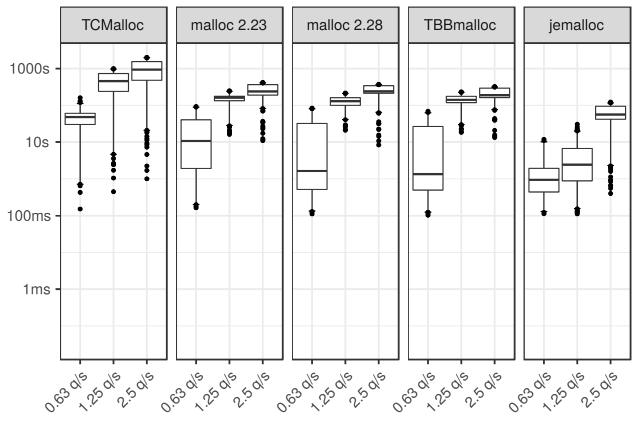


Figure 12: Query latency distributions for different query rates (4-socket Xeon, SF 100, raw allocation).

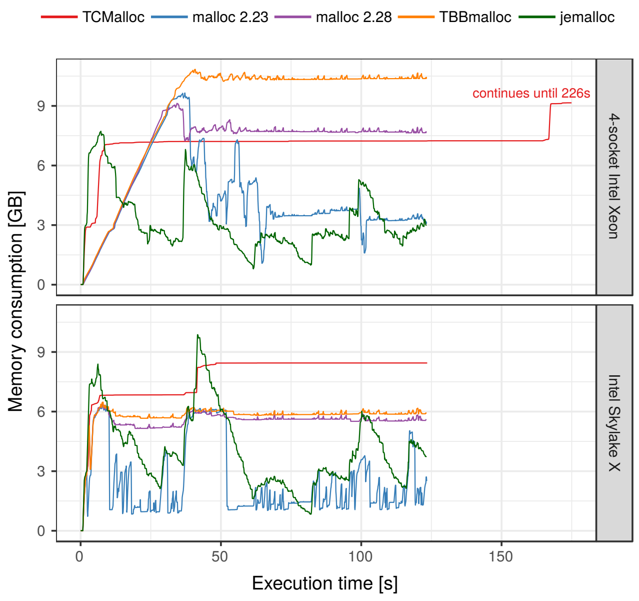


Figure 13: Memory consumption over time (λ=6 q/s, SF 10, raw allocation).

the operating system at the end of the execution. `TBBmalloc` stores small allocations in thread-local heaps which are held during the execution of the query set. The other allocators show a similar allocation pattern.

6. CONCLUSIONS

In this work, we provided a thorough experimental analysis and discussion on the impact of dynamic memory allocators for high-performance query processing. We highlighted the strength and weaknesses of the different state-of-the-art allocators according to scalability, performance, memory efficiency, and fairness to other processes. For our allocation pattern, which is probably not unlike to that of most high-performance query engines, we can summarize our findings as follows:

	scalable	fast	mem. fair	mem. efficient
TCMalloc	−−	∼	++	+
malloc 2.23	−	∼	+	∼
malloc 2.28	∼	+	−	∼
TBBmalloc	+	+	+	+
jemalloc	++	+	+	+

As a result of this work, we choose `jemalloc` as the standard allocator for our DBMS.

7. ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 725286).

8. REFERENCES

[1] R. Appuswamy, A. Anadiotis, D. Porobic, M. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of OLTP engines for high-contention workloads. *PVLDB*, 11(2):121–134, 2017.

[2] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core cpus: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[3] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48, 2011.

[4] D. Durner, V. Leis, and T. Neumann. On the impact of memory allocation on high-performance query processing. In *DaMoN*, pages 21:1–21:3, 2019.

[5] D. Durner and T. Neumann. No false negatives: Accepting all useful schedules in a fast serializable many-core system. In *ICDE*, 2019.

[6] J. Evans. Tick tock, malloc needs a clock [talk]. In *ACM Applicative*, 2015.

[7] J. Evans. jemalloc changelog. `https://github.com/jemalloc/jemalloc/blob/dev/ChangeLog`, 2018.

[8] P. Ezolt. A study in malloc: A case of excessive minor faults. In *Linux Showcase & Conference*, 2001.

[9] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *2011 12th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 92–98. IEEE, 2011.

[10] Google. Tcmalloc documentation. `https://gperftools.github.io/gperftools/tcmalloc.html`, 2007.

[11] Google. gperftools repository. `https://github.com/`

gperftools/gperftools/tree/gperftools-2.5.93, 2017.

[12] R. L. Hudson, B. Saha, A. Adl-Tabatabai, and B. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM*, pages 74–83, 2006.

[13] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[14] Intel. Threading building blocks repository. https://github.com/01org/tbb/tree/tbb_2017, 2017.

[15] A. Kemper and T. Neumann. Hyper: A hybrid oltp&olap main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[16] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[17] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[18] G. C. Library. The gnu c library version 2.26 is now available. https://sourceware.org/ml/libc-alpha/2017-08/msg00010.html, 2017.

[19] G. C. Library. Malloc internals: Overview of malloc. https://sourceware.org/glibc/wiki/MallocInternals, 2018.

[20] N. May, A. Böhm, and W. Lehner. SAP HANA - the evolution of an in-memory DBMS from pure OLAP processing towards mixed workloads. In *BTW*, pages 545–563, 2017.

[21] R. O. Nambiar and M. Poess. The making of TPC-DS. In *VLDB*, pages 1049–1058, 2006.

[22] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-memory systems. *PVLDB*, 10(11):1166–1177, 2017.

[23] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[24] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. In *DaMoN*, page 7, 2013.

[25] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[26] Z. Zhang, H. Deshmukh, and J. M. Patel. Data partitioning for in-memory systems: Myths, challenges, and opportunities. In *CIDR*, 2019.

Bibliography

[Adr22] Merv Adrian. *DBMS Market Transformation 2021: The Big Picture.* https://blogs.gartner.com/merv-adrian/2022/04/16/dbms-market-transformation-2021-the-big-picture/. accessed: 2022-09-30. 2022.

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. “Fast Algorithms for Mining Association Rules in Large Databases”. In: *VLDB*. 1994, pp. 487–499.

[AR20] Josep Aguilar-Saborit and Raghu Ramakrishnan. “POLARIS: The Distributed SQL Engine in Azure Synapse”. In: *PVLDB* 13.12 (2020), pp. 3204–3216.

[Ail+01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. “Weaving Relations for Cache Performance”. In: *VLDB*. 2001, pp. 169–180.

[Ala+12] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. “NoDB: efficient query execution on raw data files”. In: *SIGMOD*. ACM, 2012, pp. 241–252.

[All23] Alluxio. *Alluxio - Data Orchestration for the Cloud.* https://www.alluxio.io/. accessed: 2023-10-30. 2023.

[Alt+03] Mehmet Altinel, Christof Bornhövd, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. “Cache Tables: Paving the Way for an Adaptive Database Cache”. In: *VLDB*. 2003, pp. 718–729.

[Ama23a] Amazon. *Amazon EC2 On-Demand Pricing.* https://aws.amazon.com/ec2/pricing/on-demand/. accessed: 2023-10-30. 2023.

[Ama23b] Amazon. *Amazon S3 Cloud Storage.* https://aws.amazon.com/s3/. accessed: 2023-11-01. 2023.

[Ama23c] Amazon. *Amazon S3 customers.* https://aws.amazon.com/s3/customers/. accessed: 2023-10-05. 2023.

[Ama23d] Amazon. *Amazon S3 pricing.* https://aws.amazon.com/s3/pricing. accessed: 2023-11-01. 2023.

[Ama23e] Amazon. *Amazon S3 Storage Classes.* https://aws.amazon.com/s3/storage-classes/. accessed: 2023-11-01. 2023.

[Ama23f] Amazon. *AWS SDK Crt Cpp.* https://github.com/awslabs/aws-crt-cpp. accessed: 2023-11-09. 2023.

[Ama23g] Amazon. *AWS SDK for C++*. https://github.com/aws/aws-sdk-cpp. accessed: 2023-11-09. 2023.

[Ama23h] Amazon. *Spot Instance interruption notices*. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/spot-instance-termination-notices.html. accessed: 2023-11-03. 2023.

[Ama23i] Amazon Web Services. *Amazon DocumentDB*. https://aws.amazon.com/documentdb. accessed: 2023-11-03. 2023.

[Ant+19] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. "Socrates: The New SQL Server in the Cloud". In: *SIGMOD*. ACM, 2019, pp. 1743–1756.

[Apa18] Apache. *Gandiva for Apache Arrow*. https://arrow.apache.org/blog/2018/12/05/gandiva-donation/. accessed: 2023-11-03. 2018.

[Apa23a] Apache. *Apache Arrow*. https://arrow.apache.org/. accessed: 2023-11-03. 2023.

[Apa23b] Apache. *Apache Iceberg*. https://iceberg.apache.org/. accessed: 2023-11-01. 2023.

[Apa23c] Apache Software Foundation. *Apache Avro*. https://avro.apache.org/. accessed: 2023-11-03. 2023.

[Apa23d] Apache Software Foundation. *Apache Parquet*. https://parquet.apache.org. accessed: 2023-11-03. 2023.

[App+17] Raja Appuswamy, Angelos Anadiotis, Danica Porobic, Mustafa Iman, and Anastasia Ailamaki. "Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-Contention Workloads". In: *PVLDB* 11.2 (2017), pp. 121–134.

[Arm+20] Michael Armbrust, Tathagata Das, Sameer Paranjpye, Reynold Xin, Shixiong Zhu, Ali Ghodsi, Burak Yavuz, Mukul Murthy, Joseph Torres, Liwen Sun, Peter A. Boncz, Mostafa Mokhtar, Herman Van Hovell, Adrian Ionescu, Alicja Luszczak, Michal Switakowski, Takuya Ueshin, Xiao Li, Michal Szafranski, Pieter Senster, and Matei Zaharia. "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores". In: *PVLDB* 13.12 (2020), pp. 3411–3424.

[Arm+15] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD*. ACM, 2015, pp. 1383–1394.

[Arm+22] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. "Amazon Redshift Re-invented". In: *SIGMOD*. ACM, 2022, pp. 2205–2217.

[Axb19] Jens Axboe. *Efficient IO with io_uring*. `https://kernel.dk/io_uring.pdf`. accessed: 2023-11-01. 2019.

[AKA17] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. "ReCache: Reactive Caching for Fast Analytics over Heterogeneous Data". In: *PVLDB* 11.3 (2017), pp. 324–337.

[Baa+17] Mohamed Amine Baazizi, Houssem Ben Lahmar, Dario Colazzo, Giorgio Ghelli, and Carlo Sartiani. "Schema Inference for Massive JSON Datasets". In: *EDBT*. 2017, pp. 222–233.

[Bal+13] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. "Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware". In: *ICDE*. IEEE, 2013, pp. 362–373.

[BGN21] Maximilian Bandle, Jana Giceva, and Thomas Neumann. "To Partition, or Not to Partition, That is the Join Question in a Real System". In: *SIGMOD*. ACM, 2021, pp. 168–180.

[Bar18] Jeff Barr. *New C5n Instances with 100 Gbps Networking*. `https://aws.amazon.com/blogs/aws/new-c5n-instances-with-100-gbps-networking/`. accessed: 2023-11-03. 2018.

[Beh+22] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman Van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. "Photon: A Fast Query Engine for Lakehouse Systems". In: *SIGMOD*. ACM, 2022, pp. 2326–2339.

[BA22] Haoqiong Bian and Anastasia Ailamaki. “Pixels: An Efficient Column Store for Cloud Data Lakes”. In: *ICDE*. IEEE, 2022, pp. 3078–3090.

[BSA23] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. “Using Cloud Functions as Accelerator for Elastic Data Analytics”. In: *SIGMOD*. ACM, 2023, 161:1–161:27.

[BLP11] Spyros Blanas, Yinan Li, and Jignesh M. Patel. “Design and evaluation of main memory hash join algorithms for multi-core CPUs”. In: *SIGMOD*. 2011, pp. 37–48.

[Bla+14] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. “Parallel data analysis directly on scientific file formats”. In: *SIGMOD*. ACM, 2014, pp. 385–396.

[BNE13] Peter A. Boncz, Thomas Neumann, and Orri Erling. “TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark”. In: *TPCTC*. Vol. 8391. LNCS. Springer, 2013, pp. 61–76.

[BB17] Daniele Bonetta and Matthias Brantner. “FAD.js: Fast JSON Data Access Using JIT-based Speculative Optimizations”. In: *PVLDB* 10.12 (2017), pp. 1778–1789.

[Bor+04] Christof Bornhövd, Mehmet Altinel, C. Mohan, Hamid Pirahesh, and Berthold Reinwald. “Adaptive Database Caching with DB-Cache”. In: *IEEE Data Eng. Bull.* 27.2 (2004), pp. 11–18.

[BL19] Brendan Bouffler and Chris Liu. *Deep-Dive Into 100G networking & Elastic Fabric Adapter on Amazon EC2*. AWS re:Invent, `https://d1.awsstatic.com/events/reinvent/2019/REPEAT_2_Deep-dive_into_100G_networking_&_Elastic_Fabric_Adapter_on_Amazon_EC2_CMP334-R2.pdf`. accessed: 2023-11-01. 2019.

[Bra+08] Matthias Brantner, Daniela Florescu, David A. Graf, Donald Kossmann, and Tim Kraska. “Building a database on S3”. In: *SIGMOD*. ACM, 2008, pp. 251–264.

[Cai+22] Qizhe Cai, Midhul Vuppalapati, Jaehyun Hwang, Christos Kozyrakis, and Rachit Agarwal. “Towards μs tail latency and terabit ethernet: disaggregating the host network stack”. In: *SIGCOMM*. ACM, 2022, pp. 767–779.

[Can17] Luca Canali. *Performance Analysis of a CPU-Intensive Workload in Apache Spark*. `https://externaltable.blogspot.com/2017/09/performance-analysis-of-cpu-intensive.html`. Results presented at Spark Summit. accessed: 2023-11-03. 2017.

[Cao+22] Wei Cao, Feifei Li, Gui Huang, Jianghang Lou, Jianwei Zhao, Dengcheng He, Mengshi Sun, Yingqiang Zhang, Sheng Wang, Xueqiang Wu, Han Liao, Zilin Chen, Xiaojian Fang, Mo Chen, Chenghui Liang, Yanxin Luo, Huanming Wang, Songlei Wang, Zhanfeng Ma, Xinjun Yang, Xiang Peng, Yubin Ruan, Yuhui Wang, Jie Zhou, Jianying Wang, Qingda Hu, and Junbin Kang. "PolarDB-X: An Elastic Distributed Relational Database for Cloud-Native Applications". In: *ICDE*. IEEE, 2022, pp. 2859–2872.

[Cao+21] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. "PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers". In: *SIGMOD*. ACM, 2021, pp. 2477–2489.

[CLP13] Craig Chasseur, Yinan Li, and Jignesh M. Patel. "Enabling JSON Document Stores in Relational Systems". In: *WebDB*. 2013, pp. 1–6.

[CR94] Chung-Min Chen and Nick Roussopoulos. "The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching". In: *EDBT*. 1994, pp. 323–336.

[CR14] Yu Cheng and Florin Rusu. "Parallel in-situ data processing with speculative loading". In: *SIGMOD*. ACM, 2014, pp. 1287–1298.

[Cou23] Couchbase. *Couchbase Under the Hood*. `https://info.couchbase.com/rs/302-GJY-034/images/Couchbase_Under_The_Hood_WP.pdf`. accessed: 2023-11-03. 2023.

[Cry21] Crystal. *Regions workload queries*. `https://aka.ms/crystal-queries`. accessed: 2023-11-03. 2021.

[Dag+16] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. "The Snowflake Elastic Data Warehouse". In: *SIGMOD*. ACM, 2016, pp. 215–226.

[Dan+22] Jonas Dann, Royden Wagner, Daniel Ritter, Christian Faerber, and Holger Fröning. "PipeJSON: Parsing JSON at Line Speed on FPGAs". In: *DaMoN*. ACM, 2022, 3:1–3:7.

[Dar+96] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. “Semantic Data Caching and Replacement”. In: *VLDB*. 1996, pp. 330–341.

[Dat23a] Databricks. *Delta Cache*. `https://docs.databricks.com/delta/optimizations/delta-cache.html`. accessed: 2023-11-03. 2023.

[Dat23b] Databricks. *Introduction to Data Lakes*. `https://databricks.com/discover/data-lakes/introduction`. accessed: 2023-11-03. 2023.

[Des+98] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. “Caching Multidimensional Queries Using Chunks”. In: *SIGMOD*. ACM, 1998, pp. 259–270.

[Did+22] Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. “Understanding modern storage APIs: a systematic study of libaio, SPDK, and io_uring”. In: *SYSTOR*. ACM, 2022, pp. 120–127.

[Din+20] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. “Tsunami: A Learned Multi-dimensional Index for Correlated Data and Skewed Workloads”. In: *PVLDB* 14.2 (2020), pp. 74–86.

[DA16] Michael DiScala and Daniel J. Abadi. “Automatic Generation of Normalized Relational Schemas from Nested Key-Value Data”. In: *SIGMOD*. ACM, 2016, pp. 295–310.

[Dre+20] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. “Quantifying TPC-H Choke Points and Their Optimizations”. In: *PVLDB* 13.8 (2020), pp. 1206–1220.

[Dur22] Dominik Durner. *AnyBlob*. `https://github.com/durner/AnyBlob/`. accessed: 2023-10-30. 2022.

[DCL21] Dominik Durner, Badrish Chandramouli, and Yinan Li. “Crystal: A Unified Cache Storage System for Analytical Databases”. In: *PVLDB* 14.11 (2021), pp. 2432–2444.

[Dur+24] Dominik Durner, Lennart Espe, Jana Giceva, and Anja Gruenheid. “TracEx: Understanding and Analyzing Database Traces”. In: *CIDR*. 2024.

[DLN19a] Dominik Durner, Viktor Leis, and Thomas Neumann. “Experimental Study of Memory Allocation for High-Performance Query Processing”. In: *ADMS@VLDB*. 2019, pp. 1–9.

[DLN19b] Dominik Durner, Viktor Leis, and Thomas Neumann. “On the Impact of Memory Allocation on High-Performance Query Processing”. In: *DaMoN*. ACM, 2019, 21:1–21:3.

[DLN21] Dominik Durner, Viktor Leis, and Thomas Neumann. “JSON Tiles: Fast Analytics on Semi-Structured Data”. In: *SIGMOD*. ACM, 2021, pp. 445–458.

[DLN23] Dominik Durner, Viktor Leis, and Thomas Neumann. “Exploiting Cloud Object Storage for High-Performance Analytics”. In: *PVLDB* 16.11 (2023), pp. 2769–2782.

[DN19] Dominik Durner and Thomas Neumann. “No False Negatives: Accepting All Useful Schedules in a Fast Serializable Many-Core System”. In: *ICDE*. IEEE, 2019, pp. 734–745.

[Dur+17] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, and Tim Kraska. “Revisiting Reuse in Main Memory Database Systems”. In: *SIGMOD*. ACM, 2017, pp. 1275–1289.

[FN21] Philipp Fent and Thomas Neumann. “A Practical Approach to Groupjoin and Nested Aggregates”. In: *PVLDB* 14.11 (2021), pp. 2383–2396.

[Fer+11] Tais Borges Ferreira, Rivalino Matias, Autran Macedo, and Lucio Borges de Araujo. “An Experimental Study on Memory Allocators in Multicore and Multithreaded Applications”. In: *PDCAT*. IEEE, 2011, pp. 92–98.

[Fla+07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. “HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm”. In: *Discrete Mathematics & Theoretical Computer Science* (2007), pp. 137–156.

[Fre+20] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. “Adopting Worst-Case Optimal Joins in Relational Database Systems”. In: *PVLDB* 13.11 (2020), pp. 1891–1904.

[Ge+19] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. “Speculative Distributed CSV Data Parsing for Big Data Analytics”. In: *SIGMOD*. ACM, 2019, pp. 883–899.

[GL01] Jonathan Goldstein and Per-Åke Larson. “Optimizing Queries Using Materialized Views: A practical, scalable solution”. In: *SIGMOD*. ACM, 2001, pp. 331–342.

[Goo23] Google. *Cloud Storage*. `https://cloud.google.com/storage/`. accessed: 2023-11-02. 2023.

[Gra09] Goetz Graefe. “Fast loads and fast queries”. In: *DaWaK*. 2009, pp. 111–124.

[Gru+23] Ferdinand Gruber, Maximilian Bandle, Alexis Engelke, Thomas Neumann, and Jana Giceva. “Bringing Compiling Databases to RISC Architectures”. In: *PVLDB* 16.6 (2023), pp. 1222–1234.

[Gup+15] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. “Amazon Redshift and the Case for Simpler Data Warehouses”. In: *SIGMOD*. ACM, 2015, pp. 1917–1923.

[HHL20] Gabriel Haas, Michael Haubenschild, and Viktor Leis. “Exploiting Directly-Attached NVMe Arrays in DBMS”. In: *CIDR*. 2020.

[HPY00] Jiawei Han, Jian Pei, and Yiwen Yin. “Mining Frequent Patterns without Candidate Generation”. In: *SIGMOD*. ACM, 2000, pp. 1–12.

[Hel+19] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. “Serverless Computing: One Step Forward, Two Steps Back”. In: *CIDR*. 2019.

[Hug21] Jason Hughes. *Apache Iceberg: An Architectural Look Under the Covers*. `https://www.dremio.com/resources/guides/apache-iceberg-an-architectural-look-under-the-covers/`. accessed: 2023-11-03. 2021.

[Hüt+22] Thomas Hütter, Nikolaus Augsten, Christoph M. Kirsch, Michael J. Carey, and Chen Li. “JEDI: These aren’t the JSON documents you’re looking for?” In: *SIGMOD*. ACM, 2022, pp. 1584–1597.

[Idr+11] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. “Here are my Data Files. Here are my Queries. Where are my Results?” In: *CIDR*. 2011, pp. 57–68.

[Iva+09] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. “An architecture for recycling intermediates in a column-store”. In: *SIGMOD*. ACM, 2009, pp. 309–320.

[Jal+18] Virajith Jalaparti, Chris Douglas, Mainak Ghosh, Ashvin Agrawal, Avrilia Floratou, Srikanth Kandula, Ishai Menache, Joseph (Seffi) Naor, and Sriram Rao. “Netco: Cache and I/O Management for Analytics over Disaggregated Stores”. In: *SoCC*. ACM, 2018, pp. 186–198.

[Jin+18a] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. “Selecting Subexpressions to Materialize at Datacenter Scale”. In: *PVLDB* 11.7 (2018), pp. 800–812.

[Jin+18b] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. "Computation Reuse in Analytics Job Service at Microsoft". In: *SIGMOD*. ACM, 2018, pp. 191–203.

[JSO20] JSON Schema. *Specification of the new draft*. `https://json-schema.org/specification.html`. accessed: 2023-11-02. 2020.

[KAA16] Manos Karpathiotakis, Ioannis Alagiannis, and Anastasia Ailamaki. "Fast Queries Over Heterogeneous Data Through Engine Customization". In: *PVLDB* 9.12 (2016), pp. 972–983.

[Kar+17] Manos Karpathiotakis, Avrilia Floratou, Fatma Özcan, and Anastasia Ailamaki. "No data left behind: real-time insights from a complex data ecosystem". In: *SoCC*. ACM, 2017, pp. 108–120.

[KLN21] Timo Kersten, Viktor Leis, and Thomas Neumann. "Tidy Tuples and Flying Start: fast compilation and fast execution of relational queries in Umbra". In: *VLDB Journal* 30.5 (2021), pp. 883–905.

[Kli+18] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *OSDI*. USENIX, 2018, pp. 427–444.

[KLN18] André Kohn, Viktor Leis, and Thomas Neumann. "Adaptive Execution of Compiled Queries". In: *ICDE*. IEEE, 2018, pp. 197–208.

[Kor+22] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan S. Milojicic, and Gustavo Alonso. "Farview: Disaggregated Memory with Operator Off-loading for Database Engines". In: *CIDR*. 2022.

[KFD00] Donald Kossmann, Michael J. Franklin, and Gerhard Drasch. "Cache investment: integrating query optimization and distributed data placement". In: *TODS* 25.4 (2000), pp. 517–558.

[KR99] Yannis Kotidis and Nick Roussopoulos. "DynaMat: A Dynamic View Management System for Data Warehouses". In: *SIGMOD*. ACM, 1999, pp. 371–382.

[Kus+23] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. "BtrBlocks: Efficient Columnar Compression for Data Lakes". In: *SIGMOD*. ACM, 2023, 118:1–118:26.

[Lan+16] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. "Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation". In: *SIGMOD*. ACM, 2016, pp. 311–326.

[LL19] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *VLDB Journal* 28.6 (2019), pp. 941–960.

[LGZ04] Per-Åke Larson, Jonathan Goldstein, and Jingren Zhou. "MTCache: Transparent Mid-Tier Database Caching in SQL Server". In: *ICDE*. IEEE, 2004, pp. 177–188.

[Lei+23] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. "Virtual-Memory Assisted Buffer Management". In: *SIGMOD*. ACM, 2023, 7:1–7:25.

[Lei+14] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age". In: *SIGMOD*. ACM, 2014, pp. 743–754.

[Lei+15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. "How Good Are Query Optimizers, Really?" In: *PVLDB* 9.3 (2015), pp. 204–215.

[LK21] Viktor Leis and Maximilian Kuschewski. "Towards Cost-Optimal Query Processing in the Cloud". In: *PVLDB* 14.9 (2021), pp. 1606–1612.

[LB21] Alberto Lerner and Philippe Bonnet. "Not your Grandpa's SSD: The Era of Co-Designed Storage Devices". In: *SIGMOD*. ACM, 2021, pp. 2852–2858.

[Li19] Feifei Li. "Cloud native database systems at Alibaba: Opportunities and Challenges". In: *PVLDB* 12.12 (2019), pp. 2263–2272.

[Li18] Haoyuan Li. "Alluxio: A virtual distributed file system". PhD thesis. UC Berkeley, 2018.

[Li+14] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *SoCC*. ACM, 2014, 6:1–6:15.

[Li+17] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. "Mison: A Fast JSON Parser for Data Analytics". In: *PVLDB* 10.10 (2017), pp. 1118–1129.

[LLC23] Yinan Li, Jianan Lu, and Badrish Chandramouli. "Selection Pushdown in Column Stores using Bit Manipulation Instructions". In: *SIGMOD*. ACM, 2023, 178:1–178:26.

[LG15] Zhen Hua Liu and Dieter Gawlick. "Management of Flexible Schema Data in RDBMSs - Opportunities and Limitations for NoSQL -". In: *CIDR*. 2015.

[LHM14] Zhen Hua Liu, Beda Christoph Hammerschmidt, and Doug McMahon. “JSON data management: supporting schema-less development in RDBMS”. In: *SIGMOD*. ACM, 2014, pp. 1247–1258.

[Lyu+21] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. “Greenplum: A Hybrid Database for Transactional and Analytical Workloads”. In: *SIGMOD*. ACM, 2021, pp. 2530–2542.

[MBL17] Norman May, Alexander Böhm, and Wolfgang Lehner. “SAP HANA - The Evolution of an In-Memory DBMS from Pure OLAP Processing Towards Mixed Workloads”. In: *BTW*. 2017, pp. 545–563.

[Mel+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. “Dremel: Interactive Analysis of Web-Scale Datasets”. In: *PVLDB* 3.1 (2010), pp. 330–339.

[Mel+20] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. “Dremel: A Decade of Interactive SQL Analysis at Web Scale”. In: *PVLDB* 13.12 (2020), pp. 3461–3472.

[Met23] Meta. *Using the Graph API*. `https://developers.facebook.com/docs/graph-api/using-graph-api`. accessed: 2023-11-03. 2023.

[Mic23] Microsoft. *Azure Blob Storage*. `https://azure.microsoft.com/en-us/products/storage/blobs/`. accessed: 2023-11-02. 2023.

[Mil19] Tova Milo. *Getting Rid of Data*. `https://vldb2019.github.io/files/VLDB19-keynote-2-slides.pdf`. VLDB Keynote. 2019.

[Moe98] Guido Moerkotte. “Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing”. In: *VLDB*. 1998, pp. 476–487.

[Mon23a] Inc. MongoDB. *Databases vs. Data Warehouses vs. Data Lakes*. `https://www.mongodb.com/databases/data-lake-vs-data-warehouse-vs-database`. accessed: 2023-10-02. 2023.

[Mon23b] MongoDB, Inc. *MongoDB Architecture Guide*. `https://www.mongodb.com/collateral/mongodb-architecture-guide`. accessed: 2023-11-03. 2023.

[MMA20] Ingo Müller, Renato Marroquín, and Gustavo Alonso. “Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure”. In: *SIGMOD*. ACM, 2020, pp. 115–130.

[NBV13] Fabian Nagel, Peter A. Boncz, and Stratis Viglas. “Recycling in pipelined query evaluation”. In: *ICDE*. IEEE, 2013, pp. 338–349.

[NP06] Raghunath Othayoth Nambiar and Meikel Poess. “The Making of TPC-DS”. In: *VLDB*. 2006, pp. 1049–1058.

[Neu11] Thomas Neumann. “Efficiently Compiling Efficient Query Plans for Modern Hardware”. In: *PVLDB* 4.9 (2011), pp. 539–550.

[NF20] Thomas Neumann and Michael Freitag. “Umbra: A Disk-Based System with In-Memory Performance”. In: *CIDR*. 2020.

[NLK17] Thomas Neumann, Viktor Leis, and Alfons Kemper. “The Complete Story of Joins (in HyPer)”. In: *BTW*. Vol. P-265. LNI. GI, 2017, pp. 31–50.

[Ope23] OpenSSL Project Authors. *OpenSSL - Cryptography and SSL/TLS Toolkit*. `https://www.openssl.org/`. accessed: 2023-11-12. 2023.

[Ora02] Oracle. *Hardware and I/O Considerations in Data Warehouses*. `https://docs.oracle.com/cd/A97630_01/server.920/a96520/hardware.htm`. accessed: 2023-10-02. 2002.

[Ora17] Oracle. *Database Data Warehousing Guide: Using Zone Maps*. `https://docs.oracle.com/database/121/DWHSG/zone_maps.htm`. accessed: 2023-11-03. 2017.

[Pal+18] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. “Filter Before You Parse: Faster Analytics on Raw Data with Sparser”. In: *PVLDB* 11.11 (2018), pp. 1576–1589.

[Pan21] Ippokratis Pandis. “The evolution of Amazon Redshift”. In: *PVLDB* 14.12 (2021), pp. 3162–3163.

[Par+21] Jong-Hyeok Park, Soyee Choi, Gihwan Oh, and Sang Won Lee. “SaS: SSD as SQL Database System”. In: *PVLDB* 14.9 (2021), pp. 1481–1488.

[Pav+18] Christina Pavlopoulou, E. Preston Carman Jr., Till Westmann, Michael J. Carey, and Vassilis J. Tsotras. “A Parallel and Scalable Processor for JSON Data”. In: *EDBT*. 2018, pp. 576–587.

[PJ14] Luis Leopoldo Perez and Christopher M. Jermaine. “History-aware query optimization with materialized intermediate views”. In: *ICDE*. IEEE, 2014, pp. 520–531.

[Per+20] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. “Starling: A Scalable Query Engine on Cloud Functions”. In: *SIGMOD*. ACM, 2020, pp. 131–141.

[Pos23a] PostgreSQL Global Development Group. *JSON Functions and Operators*. `https://www.postgresql.org/docs/11/functions-json.html`. accessed: 2023-11-03. 2023.

[Pos23b] PostgreSQL Global Development Group. *JSON Types*. `https://www.postgresql.org/docs/11/datatype-json.html`. accessed: 2023-11-03. 2023.

[RFN23] Alice Rey, Michael Freitag, and Thomas Neumann. “Seamless Integration of Parquet Files into Data Processing”. In: *BTW*. Vol. P-331. LNI. GI, 2023, pp. 235–258.

[SEZ23] Majid Saeedan, Ahmed Eldawy, and Zhijia Zhao. “dsJSON: A Distributed SQL JSON Processor”. In: *SIGMOD*. ACM, 2023, 103:1–103:25.

[SDQ10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. “Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance”. In: *PVLDB* 3.1 (2010), pp. 460–471.

[SSV96] Peter Scheuermann, Junho Shim, and Radek Vingralek. “WATCHMAN : A Data Warehouse Intelligent Cache Manager”. In: *VLDB*. 1996, pp. 51–62.

[Set+19] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. “Presto: SQL on Everything”. In: *ICDE*. IEEE, 2019, pp. 1802–1813.

[SI17] Supreeth Shastri and David E. Irwin. “HotSpot: automated server hopping in cloud spot markets”. In: *SoCC*. ACM, 2017, pp. 493–505.

[SSV99] Junho Shim, Peter Scheuermann, and Radek Vingralek. “Dynamic Caching of Query Results for Decision Support Systems”. In: *SSDBM*. 1999, pp. 254–263.

[SDN98] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. “Materialized View Selection for Multidimensional Datasets”. In: *VLDB*. 1998, pp. 488–499.

[SN22] Moritz Sichert and Thomas Neumann. “User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases”. In: *PVLDB* 15.5 (2022), pp. 1119–1131.

[Sri+96] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. "Answering Queries with Aggregation Using Views". In: *VLDB*. 1996, pp. 318–329.

[SA13] Radu Stoica and Anastasia Ailamaki. "Enabling efficient OS paging for main-memory OLTP databases". In: *DaMoN*. ACM, 2013, p. 7.

[Sto+90] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. "On Rules, Procedures, Caching and Views in Data Base Systems". In: *SIGMOD*. ACM, 1990, pp. 281–290.

[Sub+15] Supreeth Subramanya, Tian Guo, Prateek Sharma, David E. Irwin, and Prashant J. Shenoy. "SpotOn: a batch computing service for the spot market". In: *SoCC*. ACM, 2015, pp. 329–341.

[Sun+14] Liwen Sun, Michael J. Franklin, Sanjay Krishnan, and Reynold S. Xin. "Fine-grained partitioning for aggressive data skipping". In: *SIGMOD*. ACM, 2014, pp. 1115–1126.

[Tab23] Tableau. *Hyper API*. `https://www.tableau.com/developer/tools/hyper-api`. accessed: 2023-11-03. 2023.

[TDA14] Daniel Tahara, Thaddeus Diamond, and Daniel J. Abadi. "Sinew: a SQL system for multi-structured data". In: *SIGMOD*. ACM, 2014, pp. 815–826.

[Tan+19a] Junjay Tan, Thanaa M. Ghanem, Matthew Perron, Xiangyao Yu, Michael Stonebraker, David J. DeWitt, Marco Serafini, Ashraf Aboulnaga, and Tim Kraska. "Choosing A Cloud DBMS: Architectures and Tradeoffs". In: *PVLDB* 12.12 (2019), pp. 2170–2182.

[TGO01] Kian-Lee Tan, Shen-Tat Goh, and Beng Chin Ooi. "Cache-on-Demand: Recycling with Certainty". In: *ICDE*. IEEE, 2001, pp. 633–640.

[Tan+19b] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. "Intermittent Query Processing". In: *PVLDB* 12.11 (2019), pp. 1427–1441.

[Thu+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. "Hive - A Warehousing Solution Over a Map-Reduce Framework". In: *PVLDB* 2.2 (2009), pp. 1626–1629.

[Thu+10] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Anthony, Hao Liu, and Raghotham Murthy. "Hive - a petabyte scale data warehouse using Hadoop". In: *ICDE*. IEEE, 2010, pp. 996–1005.

[Tu+13] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. “Speedy transactions in multicore in-memory databases”. In: *SOSP*. ACM, 2013, pp. 18–32.

[Van+18] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. “Eon Mode: Bringing the Vertica Columnar Database to the Cloud”. In: *SIGMOD*. ACM, 2018, pp. 797–809.

[Ver+17] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. “Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases”. In: *SIGMOD*. ACM, 2017, pp. 1041–1052.

[VMw23] VMware. *Greenplum Database - Massively Parallel Postgres for Analytics*. `https://www.greenplum.org/`. accessed: 2023-11-03. 2023.

[Vup+20] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. “Building An Elastic Query Engine on Disaggregated Storage”. In: *NSDI*. USENIX, 2020, pp. 449–462.

[Wan+22] Ruihong Wang, Jianguo Wang, Stratos Idreos, M. Tamer Özsu, and Walid G. Aref. “The Case for Distributed Shared-Memory Databases with RDMA-Enabled Memory Disaggregation”. In: *PVLDB* 16.1 (2022), pp. 15–22.

[Win+22] Christian Winter, Jana Giceva, Thomas Neumann, and Alfons Kemper. “On-Demand State Separation for Cloud Data Warehousing”. In: *PVLDB* 15.11 (2022), pp. 2966–2979.

[Win+20] Christian Winter, Tobias Schmidt, Thomas Neumann, and Alfons Kemper. “Meet Me Halfway: Split Maintenance of Continuous Views”. In: *PVLDB* 13.11 (2020), pp. 2620–2633.

[X C23] X Corp. *Twitter OpenAPI*. `https://api.twitter.com/2/openapi.json`. accessed: 2023-11-03. 2023.

[Xie+19] Dong Xie, Badrish Chandramouli, Yinan Li, and Donald Kossmann. “FishStore: Faster Ingestion with Subset Hashing”. In: *SIGMOD*. ACM, 2019, pp. 1711–1728.

[YLH23] Cong Yan, Yin Lin, and Yeye He. “Predicate Pushdown for Data Science Pipelines”. In: *SIGMOD*. ACM, 2023, 136:1–136:28.

[Yan+21] Yifei Yang, Matt Youill, Matthew E. Woicik, Yizhou Liu, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. "FlexPushdownDB: Hybrid Pushdown and Caching in a Cloud DBMS". In: *PVLDB* 14.11 (2021), pp. 2101–2113.

[Yel23] Yelp. *Yelp Dataset Challenge*. `https://www.yelp.com/dataset`. accessed: 2023-11-03. 2023.

[Yu+20] Xiangyao Yu, Matt Youill, Matthew E. Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. "PushdownDB: Accelerating a DBMS Using S3 Computation". In: *ICDE*. IEEE, 2020, pp. 1802–1805.

[Zah+16] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. "Apache Spark: a unified engine for big data processing". In: *Commun. ACM* 59.11 (2016), pp. 56–65.

[Zen+23] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. "An Empirical Evaluation of Columnar Storage Formats". In: *CoRR* abs/2304.05028 (2023).

[Zha+19] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. "AnalyticDB: Real-time OLAP Database System at Alibaba Cloud". In: *PVLDB* 12.12 (2019), pp. 2059–2070.

[Zha+22] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. "CompuCache: Remote Computable Caching using Spot VMs". In: *CIDR*. 2022.

[Zha+20] Qizhen Zhang, Yifan Cai, Sebastian Angel, Vincent Liu, Ang Chen, and Boon Thau Loo. "Rethinking Data Management Systems for Disaggregated Data Centers". In: *CIDR*. 2020.

[Zha+21] Yingqiang Zhang, Chaoyi Ruan, Cheng Li, Jimmy Yang, Wei Cao, Feifei Li, Bo Wang, Jing Fang, Yuhui Wang, Jingze Huo, and Chao Bi. "Towards Cost-Effective and Elastic Cloud Database Deployment via Memory Disaggregation". In: *PVLDB* 14.10 (2021), pp. 1900–1912.

[ZDP19] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. "Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities". In: *CIDR*. 2019.

[Zho+07] Jingren Zhou, Per-Åke Larson, Jonathan Goldstein, and Luping Ding. "Dynamic Materialized Views". In: *ICDE*. IEEE, 2007, pp. 526–535.

[ZBL22] Tobias Ziegler, Carsten Binnig, and Viktor Leis. "ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA". In: *SIGMOD*. ACM, 2022, pp. 685–699.