Rapid Data Ingestion through DB-OS Co-design

KYUNGMIN LIM[*], Seoul National University, Republic of Korea
MINSEOK YOON[*], Hanyang University, Republic of Korea
KIHWAN KIM[*], Hanyang University, Republic of Korea
ALAN DAVID FEKETE, University of Sydney, Australia
HYUNGSOO JUNG[†], Seoul National University, Republic of Korea

Sequential data access for the rapid ingestion of large fact tables from storage is a pivotal yet resource-intensive operation in data warehouse systems, consuming substantial CPU cycles across various components of DBMSs and operating systems. Although bypassing these layers can eliminate access latency, concurrent access to the same table often results in redundant data fetching due to cache-bypassing data transfers. Thus, a new design for data access control is necessary to enhance rapid data ingestion in databases. To address this concern, we propose a novel DB-OS co-design that efficiently supports sequential data access at full device speed. Our approach, zicIO, liberates DBMSs from data access control by preparing required data just before DBMSs access it, while alleviating all known I/O latencies. The core of zicIO lies in its DB-OS co-design, which aims to (1) *automate* data access control and (2) *relieve* redundant data fetching through seamless collaboration between the DB and the OS. We implemented zicIO and integrated it with four databases to demonstrate its general applicability. The evaluation showed performance enhancements of up to 9.95× under TPC-H loads.

CCS Concepts: • **Information systems → DBMS engine architectures**; **Data scans**.

Additional Key Words and Phrases: DB-OS Co-design, automated I/O control, OS-level page sharing

ACM Reference Format:
Kyungmin Lim, Minseok Yoon, Kihwan Kim, Alan David Fekete, and Hyungsoo Jung. 2025. Rapid Data Ingestion through DB-OS Co-design. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 68 (February 2025), 28 pages. https://doi.org/10.1145/3709718

1 Introduction

Sequential data access is essential in data warehouse systems, where the speed of ingesting large fact tables impacts query performance. Despite advancements such as zero-copy I/O, achieving optimal execution remains elusive due to the significant CPU cycles consumed by data access control mechanisms within the DBMS and the operating system. From a holistic standpoint, data access latency stems from five main factors: 1 *DBMS buffer cache* (**L1**), 2 *DBMS storage manager (mode switch)* (**L2**), 3 *data copy* (**L3**), 4 *OS storage stacks* (**L4**), and 5 *physical I/O latency* (**L5**). Figure 1 shows the latency breakdown for four DBMSs running the TPC-H query (Q6) on 50–100 GiB of the `LINEITEM` table, with and without Linux *readahead* [30]. The results highlight that data

[*]These authors contributed equally to this research.
[†]Corresponding author.

Authors' Contact Information: Kyungmin Lim, kyungmin.lim@snu.ac.kr, Seoul National University, Seoul, Republic of Korea; Minseok Yoon, minseok0yoon@hanyang.ac.kr, Hanyang University, Seoul, Republic of Korea; Kihwan Kim, kihwan@hanyang.ac.kr, Hanyang University, Seoul, Republic of Korea; Alan David Fekete, alan.fekete@sydney.edu.au, University of Sydney, Sydney, Australia; Hyungsoo Jung, Seoul National University, Seoul, Republic of Korea, hyungsoo.jun g@snu.ac.kr.

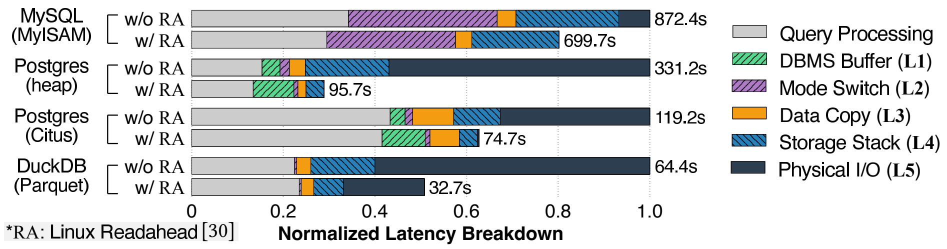


Fig. 1. Latency breakdown for Q6 in TPC-H (cold access).

access latency is critical to query performance. While bypassing all the layers can theoretically eliminate the known sources of latency, queries that access the same table undergo *redundant data fetching*, as bypassing crucial cache layers negates the benefits of caching. Addressing these challenges is complex but promises better query performance.

The OS community has made notable progress in mitigating access latency within OS components. Prior studies [6, 15, 45, 63] have focused on two key areas: (i) the mode switch (**L2**) between user mode and kernel mode for privileged device access, and (ii) the I/O delays (**L4**) within Linux storage stacks. These efforts relieve overhead by enabling applications to directly access raw storage devices (e.g., using SPDK) through the *user-level execution of custom-built I/O stacks*. While the OS-level solutions successfully lower latency by shifting I/O management to the database layer, they pose additional complexity, requiring DBMSs to develop and maintain custom I/O stacks. This shift not only leaves unresolved sources of latency within DBMSs but also imposes new burdens for user-level direct I/O control on already-overloaded database systems.

The database community has also invested substantial effort in optimizing DBMS engines to hide I/O latencies (**L4** & **L5**) by exploiting asynchronous I/O interfaces (e.g., `io_uring` [27, 28]). For instance, I/O-optimized LeanStore [17] adopted a highly parallelized architecture to maximize system efficiency. AnyBlob [14] leveraged `io_uring` to reduce CPU overhead, while Merzljak et al. [59] proposed efficient scans with coroutines. More recently, MosaicDB [22] has adopted stackless coroutines and sophisticated scheduling policies to mask latency, including I/O delays. Despite these advancements, database optimizations remain constrained by the unavoidable CPU overhead within OS storage stacks, thus offloading some of this burden to auxiliary resources such as OS threads. To overcome these limitations, emerging research [36, 37, 50] has taken a more radical approach by exploring DB-OS co-design, developing DBMS-optimized operating systems. Inspired by these efforts, we aim to further advance DB-OS co-design to enhance rapid data ingestion, targeting both performance and system efficiency.

Despite numerous strategies for optimizing latency on both sides, the CPU overhead associated with managing data access, particularly I/O control, raises a fundamental question: who should manage data access control? The ideal answer is, *whichever can most effectively tame the five latency factors*. To achieve this, we propose a **DB-OS co-design that automates data access control**. This idea shifts the responsibility for data access management from DBMSs to a DBMS-oriented OS (DBOS), specially co-designed to *prefetch data just before the DBMS requires it*.

Three essential components are necessary to realize this vision: (i) **precise timing information from DBMSs**, indicating when specific data will be needed; (ii) **control automation at the OS level**, ensuring that I/O requests are issued directly and in a timely manner to storage devices; (iii) **seamless coordination between the OS and the DBMS** through DB-OS co-design, preventing any friction between these layers. Moreover, a **careful co-design for data-sharing mechanisms** is essential to reduce redundant data fetching when concurrent queries scan the same table. This collaborative DB-OS approach aims to eliminate overheads, thus enabling rapid data ingestion and improving overall query performance.

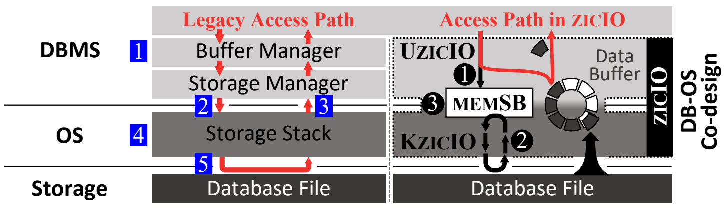


Fig. 2. Data access paths: legacy vs. ZICIO.

We present a novel DB-OS co-design, ZICIO (zero-interaction & copy I/O) that empowers DBMSs to access data at full device speed. Aligned with the three key requirements, ZICIO comprises three components: ❶ **UZICIO (user library)** that gathers precise timing information from DBMSs to predict data needs; ❷ **KZICIO (OS module)** that automates control and directly issues I/O requests to storage devices; ❸ **MEMSB (shared memory)**, small memory mapped to both the DBMS and the OS, ensuring seamless coordination between UZICIO and KZICIO. In ZICIO, DBMSs access data via UZICIO that measures the DBMS ingestion speed, updates the metric in MEMSB, and returns pages to the DBMS. Hence, UZICIO is the primary data gateway, liberating DBMSs from the overhead of the first three latency factors (i.e., 1-3). Figure 2 illustrates a new data access path in ZICIO compared to the legacy access path.

While UZICIO serves DBMSs for efficient data ingestion, KZICIO automates I/O control to ensure on-time pre-fetching of data using *μs-scale software timers* [4, 5], which tick during trigger events, such as I/O interrupt handlers. These self-clocking software timers (*soft timers*) activate our control logic that continuously monitors MEMSB and dispatches I/O requests without delay. This automated I/O management offloads CPU cycles, significantly reducing the overhead associated with 4 (OS storage stacks) and 5 (physical I/O latency). Finally, we propose SKZICIO (*sharing-enabled* KZICIO), a co-designed mechanism that alleviates redundant data fetching when DBMSs ingest data from the same tables. The core mechanisms of SKZICIO include: (i) **OS-level sliding-window cache** that temporarily holds file data within a limited number of OS pages to efficiently support concurrent scans; (ii) **dynamic page table manipulation** that dynamically pre-maps and unmaps shareable OS pages to/from the page tables of DBMS worker threads, enabling seamless and non-redundant access across multiple queries.

To accommodate columnar engines accessing multiple column groups sequentially, we modified DBMS internals—specifically range scan routines—to supply UZICIO with a complete list of required ranges, effectively transforming disjoint range scans into a single, continuous page-access stream. We implemented ZICIO as both a user library and an operating system component on Linux 5.15 with the `ext4` file system. For evaluation, we tested four different engines—MySQL (MyISAM), PostgreSQL, Citus [10] (a distributed columnar extension for PostgreSQL)), and DuckDB [47] (a columnar engine using Parquet [3])—and integrated UZICIO into their respective storage managers. We ran the TPC-H benchmark at scale factor (SF) 100, producing dataset sizes of roughly 119 to 179 GiB, and configured our server with 16 to 256 GiB of physical memory. By eliminating the need for direct interaction with the Linux kernel, ZICIO allows all of these DBMSs to concentrate on data ingestion. As a result, the four database systems using ZICIO outperform their vanilla counterparts by up to 9.95×, achieving performance levels comparable to configurations with sufficient memory (i.e., 256 GiB, exceeding the dataset size).

We make the following contributions.

- We propose a novel DB-OS co-design (ZICIO) that liberates DBMSs from data access control and enables them to focus on data ingestion with all known I/O latencies alleviated.
- Our co-design enables automated I/O control that prepares data before DBMS workers access it, with minimal CPU cycles.

- Our co-design for page sharing enables DBMS workers to ingest data from the same table with reduced I/O redundancy.
- Evaluation results show that zicIO enables DBMSs to rapidly ingest huge files, outpacing vanilla systems by a wide margin.

2 Background and Motivation

Data access delays, primarily caused by I/O control overhead, have gained prominence in databases with the advent of fast storage devices. As illustrated earlier in Figure 1, the sequential data access required by the analytic query consumes a substantial portion of execution time across both DBMS and OS components. The database and OS communities have made immense efforts to optimize inefficient I/O paths, leading to proposals that often offload significant portions of the OS storage stack to already-busy applications. The database community has adopted advanced techniques (e.g., coroutines) to hide latencies, relying on auxiliary helpers to mitigate overhead. However, our analysis of prior work reveals that the responsibility for I/O control often shifts back and forth between DBMSs and OSs, leaving overall costs largely unchanged. In the following sections, we elaborate on these observations and explore potential solutions for tackling these persistent challenges through more efficient DB-OS collaboration.

2.1 Pure OS- and DBMS-level Designs

2.1.1 ***Limitations of pure OS-level approaches.*** Over a decade, numerous studies [6–8, 15, 24, 39, 44, 45, 61–63] have explored ways to mitigate OS latency in storage and network stacks. The prevailing design trend is the disaggregation of data and control plane paths in OS I/O stacks. This approach integrates library OSs with applications for fast data transfer while employing lightweight control plane OS to manage memory access between the OS and applications. While this application-oblivious strategy is effective for general-purpose OSs, DB-OS co-design that exploits database semantics can further optimize latency factors specific to DBMSs. OS-level solutions, however, require adaptation of database storage logic for integrating library OSs with DBMSs, which is daunting for database vendors. Even adopting relatively mature OS features, such as POSIX asynchronous I/O (`aio`) [54] or Linux `io_uring` [27, 28], often needs major system overhauls [40, 43]. Thus, ensuring user-friendliness in the design of new I/O features is essential for seamless adoption by database vendors.

2.1.2 ***Limitations of pure DBMS-level approaches.*** Our community has developed state-of-the-art latency-hiding techniques. Specifically, recent studies [14, 22, 59] have proposed the idea of encapsulating potential blocking points into coroutines, which are executed through separate flows to effectively mask latency. However, a critical aspect is the management of execution logic for reading data from storage, which we term I/O control overhead. Database solutions offload this overhead to dedicated workers (e.g., using `io_uring` kernel threads), while the OS community provides library OSs, allowing applications to directly manage I/O operations. This shifting of responsibility between DBMSs and OSs reveals a fundamental gap—neither fully resolves the I/O control burden, leaving inefficiencies in both pure OS- and DBMS-level designs. To address these challenges, we argue for a novel DBOS design that separates I/O control from DBMSs and performs it asynchronously with minimal CPU overhead. By delegating I/O control to a co-designed OS tailored to the needs of the DBMS, this approach can unlock the full potential of high-speed storage devices, reducing latency and enhancing query performance and resource utilization.

2.2 DB-OS Co-design for Rapid Data Ingestion

2.2.1 ***Rationale for DB-OS co-design.*** Recent proposals [36, 37] exploring DBMS-optimized OS designs have inspired us to approach data access control through DB-OS co-design. Achieving this goal requires identifying the essential mechanisms for automating data access control. Specifically, three critical mechanisms are necessary: (i) **precise timing information** provided by the DBMS, which is best suited to provide accurate details about when and which data is needed; (ii) **timely release of I/O requests** managed by the OS, leveraging lightweight mechanisms to issue I/O operations at precise intervals, ensuring data is ready just in time; (iii) **seamless coordination between the DBMS and the OS**, enabled through a small, shared memory area mapped to both, allowing real-time metadata sharing for swift collaboration.

In zicIO, these three requirements are addressed by the design of three core components: UzicIO, KzicIO, and memSB. Each component plays a distinct role in harmonizing the strengths of both the DBMS and the OS. UzicIO serves as the primary data gateway, managing DMA-able buffer memory to facilitate zero-copy data access for DBMS workers. It also calculates DBMS ingestion speed based on data consumption patterns. KzicIO ensures precise I/O control by executing timely operations at the OS level, using soft timers to issue I/O requests without delays. memSB bridges the two components, enabling seamless communication and coordination between the DBMS and the OS by maintaining shared metadata.

2.2.2 ***How does zicIO address the five latency factors?*** The three components of zicIO effectively mask latency and save CPU cycles across the five latency factors. Since UzicIO conceals all latency factors by serving DBMS workers, DBMSs can earn zero-delay data access—unless the underlying storage device becomes saturated. The remaining latency factors are mitigated by the specialized mechanisms within zicIO as follows:

(1) ***DBMS buffer*** **(L1)**. zicIO replaces DBMS *shared* buffers with *private* ring buffers, eliminating the associated overhead.
(2) ***Frequent mode switches*** **(L2)**. By *offloading* I/O handling to KzicIO, zicIO eliminates the need for frequent mode switches between user and kernel space for I/O system calls, significantly reducing CPU cycle consumption.
(3) ***Data copy overhead*** **(L3)**. zicIO leverages *DMA-able memory* to enable zero-copy I/O between DBMSs and storage devices, avoiding unnecessary data copies and the relevant CPU cycles.
(4) ***Latencies in OS storage stacks*** **(L4)**. Traditional file systems (FS) respond to DBMS data requests by providing either cached pages or block addresses for physical I/O. zicIO *bypasses* these conventional stacks by using KzicIO to *directly issue I/O requests to devices* with prefetched logical block addresses (LBAs), minimizing CPU cycles spent in storage stack operations.
(5) ***Physical I/O delay*** **(L5)**. zicIO *hides* physical I/O delays by *timely prefetching* data when needed. To this end, DBMSs provide timing information via UzicIO, enabling KzicIO to perform prefetching to ensure that data is available just in time, effectively masking physical I/O delays.

2.3 DB-OS Co-design for OS-level Data Sharing

2.3.1 ***Motivation.*** When data warehouse queries accessing a few fact tables are executed concurrently, any approaches that bypass many layers in DBMSs and OSs to eliminate I/O latencies will encounter redundant data fetching due to the absence of caching layers. Such redundant I/O may quickly saturate device bandwidth as concurrency increases. We need to alleviate this issue without reverting to legacy shared cache designs while retaining data access efficiency. This motivation has led us to devise the concept of data sharing at the OS level, which necessitates a novel DB-OS co-design for implementing memory sharing at the OS level. To achieve OS-level memory sharing, we require two core components:

(1) ***A shared pool of OS memory pages.*** This shared pool functions as a sliding-window cache that holds file chunk data.
(2) ***Dynamic page table manipulation.*** It maps and unmaps OS pages to/from DBMS page tables for OS-level page sharing.

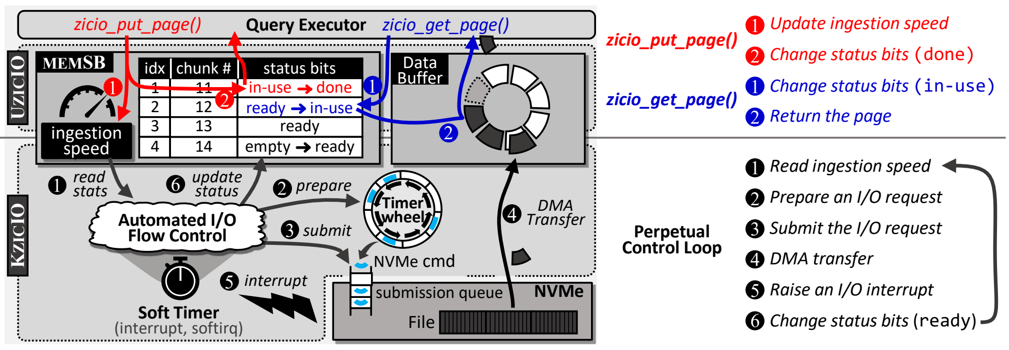


Fig. 3. The overall designs and operational aspects of zicIO.

The primary objective of OS-level data sharing is to enable DBMS workers to share as much data as possible *without explicit control*.

2.3.2 ***DB-OS co-design for OS-level page sharing.*** We have developed an OS-level page-sharing mechanism, SKzicIO (i.e., *sharing-enabled* KzicIO). The primary insight is using a shared pool of a few OS pages into which file chunks from storage are loaded. Consequently, this shared pool works as a *best-effort sliding-window cache* for an incoming stream of file chunks. Once OS pages in the pool become shareable, SKzicIO promptly *pre-maps* these OS pages to the page table of its DBMS worker, ensuring their visibility to the DBMS. After the DBMS ingests data, SKzicIO *unmaps* the pages from the database's page table for recycling. Hence, SKzicIO promises data sharing for DBMS workers without explicit control, thereby reducing the I/O redundancy.

3 RAPID DATA INGESTION ARCHITECTURE

We present an overall architecture for our DB-OS co-design, with the primary objective of decoupling data access control from DBMS workers and ensuring timely data delivery. This architecture encompasses the system designs and operational logic for three key parts: memSB, UzicIO, and KzicIO. To enable efficient operation, DBMS workers create and utilize their own zicIO channels.

3.1 Overview of zicIO

UzicIO (user library). UzicIO is a user-level library that collaborates with KzicIO for on-time data delivery. It collects data access patterns from DBMS workers, estimates precise timing information, and shares these metrics with KzicIO through memSB, thus ensuring that data pages are prefetched and ready just before they are needed by DBMS workers. The key functions of UzicIO include: (i) gathering access patterns and metadata, (ii) data gateway (APIs) for DBMS workers, and (iii) updating metric information in memSB.

***memSB* (shared memory).** memSB is shared memory created for each zicIO channel and mapped to both the DBMS and the OS, bridging our UzicIO and KzicIO components. It maintains metadata to coordinate I/O control efficiently. The key aspects of memSB include: (i) maintaining data ingestion metrics, (ii) fast communication medium for UzicIO and KzicIO, (iii) book-keeping page-sharing status for seamless collaboration between UzicIO and SKzicIO.

***KzicIO* (OS module).** KzicIO is the OS component that is responsible for automating I/O control based on the information provided by UzicIO through memSB. Its primary functions include: (i) operating soft timers to activate I/O control logic, (ii) monitoring memSB for timely prefetching, and (iii) file system-bypassing submission of I/O requests to storage devices.

Figure 3 illustrates the overall designs and operational aspects of UzicIO, KzicIO, and memSB. The next section provides detailed insights into the individual responsibilities and interactions of these components, stressing how their collaboration reduces latency.

3.2 memSB: Bridging UzicIO and KzicIO

The memory switchboard (memSB) is a small memory region accessible by both the DBMS and the OS, acting as an intermediary between UzicIO and KzicIO. memSB consists of two key elements: **metric variables** for I/O control and a **buffer status table** to track data access and availability.

Metric variables. Metric variables maintain important metrics related to DBMS data ingestion. When DBMS workers invoke UzicIO APIs, these variables are updated by UzicIO to reflect the current status of data ingestion. Meanwhile, KzicIO monitors these metrics to perform timely prefetching, ensuring that data is available when needed without delays.

Buffer status table. The buffer status table tracks the state of buffer frames to manage the flow of data. Each entry corresponds to a buffer frame and contains a file chunk ID along with a 2-bit status field that indicates one of the following four states:

(1) `empty`: The buffer frame is unused and may not have a valid OS page mapping. This state indicates that it is available for data to be prefetched by KzicIO.
(2) `ready`: KzicIO has prefetched data into the frame, and it is now marked as ready for DBMS workers to use through UzicIO.
(3) `in-use`: The frame is currently being accessed or processed by a DBMS worker. This state prevents KzicIO from mistakenly accessing this frame for prefetching.
(4) `done`: The DBMS has finished processing the data in the frame. KzicIO can now recycle the frame by marking it as `empty`.

By maintaining these state transitions, memSB manages the life cycle of buffer frames and ensures that data flows smoothly between DBMSs and storage devices without unnecessary delays.

3.3 UzicIO: The Only Data Gateway

UzicIO acts as the exclusive data gateway between the DBMS and storage, ensuring that DBMS workers can ingest data pages without experiencing access latency.

3.3.1 ***Initialization and setup***. A DBMS worker creates a zicIO channel through UzicIO APIs, specifying a file descriptor set for the target files it wishes to ingest and optionally a list of file ranges if disjoint sequential access is required for columnar engines (e.g., Citus [10] and DuckDB [47]). Three parts require initialization:

(1) ***Memory initialization***. The UzicIO library allocates the necessary buffer memory and passes it to our internal system call, `zicio_sys_open()`. It makes this buffer space DMA-able, enabling zero-copy I/O operations. Additionally, it creates memSB associated with the data buffers to collect various statistics. This memSB is then mapped to the DBMS and the OS, allowing both to see and access the shared metadata.
(2) ***I/O initialization***. `zicio_sys_open()` initializes I/O control for KzicIO by fetching the *extent tree blocks* (i.e., file `inode`) that contain LBAs for data blocks in the target database file. It then stores these LBAs into *address arrays* within the zicIO descriptor. This *pre-filled address array* allows KzicIO to initiate file system-bypassing submission of I/O requests.
(3) ***Starting the self-clocking control loop***. Finally, our system call sends the initial set of I/O requests to storage devices to prefetch data pages from target database files and start the measurement. These I/O requests trigger KzicIO's self-clocking control loop by activating soft timers through I/O interrupts. This activity also initializes memSB, which contains `null` metric data right after a zicIO channel is created.

3.3.2 ***Memory management***. UzicIO manages two memory regions to facilitate efficient data access and metadata handling: (i) DMA-able memory (data buffers) and (ii) memSB for keeping metadata. These memory regions are shared between the DBMS and the OS, enabling UzicIO and KzicIO to collaborate for data transfer.

- ***DMA-able memory (data buffers)***. We allocate a configurable region of DMA-capable memory to store data that both the DBMS and the operating system can directly access—enabling true zero-copy I/O. Specifically, we reserve 32 MiB of physical memory divided into sixteen 2 MiB huge pages, which is the minimum capacity required to accommodate our 7 GiB/s NVMe SSD. Although zicIO supports smaller 4 KiB pages, using larger 2 MiB pages reduces the overhead of managing numerous small pages.
- ***Memory management***. To facilitate quick lookups and efficient control, we paginate and index our data buffer. Each page's state is tracked with status bits, capturing one of four possible conditions (see Section 3.2). This design simplifies monitoring and streamlines memory management.
- ***Memory access***. DBMS workers request data pages through UzicIO APIs and access them directly via virtual address pointers—without any mode switches. While UzicIO handles buffer memory in units of 2 MiB huge pages, it still allows DBMS workers to operate at different page sizes (such as 8 KiB or 2 MiB) for disjoint sequential access, which is particularly useful in columnar database engines that internally exploit disjoint sequential access.

3.3.3 ***Gathering metadata information***. Upon receiving a data access request from the DBMS worker, UzicIO updates memSB with three key pieces of information: (i) the current access time, which serves as the basis for calculating per-page data ingestion rates, (ii) the buffer states, representing the status of each data page, and (iii) additional metadata for coordination with KzicIO.

3.3.4 ***Operational workflow***. We provide a detailed explanation of the operational workflow of zicIO channels, illustrating how DBMS workers interact with UzicIO that collaborates with KzicIO for supporting rapid data ingestion (**labels refer to Figure 3**):

(1) ***Data access and management***. When a DBMS worker needs data, it invokes the relevant UzicIO API (i.e., `zicio_get_page()`) to obtain a pointer to the requested data. UzicIO leverages `rdtsc` [26] to record a timestamp only when accessing a 2 MiB huge page for the first time; subsequent accesses at sub-page granularity do not trigger further timestamp recording.

(2) ***Data retrieval and delivery***. UzicIO retrieves the ready pages from the buffer memory, generally in any order, but in sequence if range scans are requested. It changes the status bits of these pages from '`ready`' to '`in-use`' (❶). The data pages are then sent to the DBMS worker, ensuring the data is immediately available without additional processing delays (❷).

(3) ***Page return and ingestion speed calculation***. After processing, the DBMS worker returns the 2 MiB page through the UzicIO API (i.e., `zicio_put_page()`). Upon return, UzicIO captures a timestamp using `rdtsc` and compares it with the timestamp recorded during the page's first access for calculating the ingestion speed for the 2 MiB page accurately (❶). It then updates the buffer state of the returned pages to '`done`' in memSB (❷), indicating that these memory spaces are free for reuse by KzicIO.

3.4 KzicIO: Perpetual Control Loop

This section details the architecture of KzicIO, designed to ensure on-time data delivery to buffer memory. The primary control mechanism focuses on *automating the data prefetching process while avoiding bursty I/O traffic*, necessitating sophisticated rate control. As illustrated in Figure 3, KzicIO collaborates seamlessly with UzicIO through memSB to execute its perpetual control loop effectively.

3.4.1 ***Requirements for automated I/O control***. KzicIO is designed to eliminate latency within the storage stack (**L4**) and mask physical I/O delays (**L5**) by bypassing the traditional file system (FS) and submitting I/O requests directly to devices in a timely manner. To achieve this, KzicIO must satisfy several crucial requirements:

- ***Precise timing information for scheduling I/O requests***. KzicIO depends on accurate timing information to initiate I/O requests exactly when data is needed. It uses the data ingestion speed—calculated and updated by UzicIO within memSB—to forecast when a DBMS worker will access specific data. For sequential scans, the access pattern is generally predictable by default. For range scans, DBMS workers provide explicit future access details through UzicIO APIs, which are then recorded in memSB.

- ***High-resolution timers for on-time issuing of I/O requests***. Achieving precise I/O scheduling demands high-resolution timers to continuously track memSB and issue I/O requests with minimal latency. To accomplish this, KzicIO employs self-clocking soft timers [4, 5], providing microsecond-level accuracy. We adapted the Linux timer wheel structure [51] to manage these scheduled I/O requests, ensuring they are dispatched exactly when needed.
- ***Preloading LBAs to bypass file systems***. To dispatch I/O requests directly to storage devices, KzicIO needs to know the LBAs of the data pages in question. It obtains this information by retrieving the appropriate `inode` and preparing the corresponding I/O requests for the data blocks. Because each zicIO channel provides only limited space for memSB, KzicIO optimizes metadata ingestion by interleaving `inode` requests with normal data requests and immediately issuing them to the storage devices.

3.4.2 ***Perpetual control loop for handling I/O requests***. Here, we describe the automated control loop that governs how I/O requests are scheduled and issued for prompt data delivery (**labels correspond to Figure 3**).

(1) ***Initialization***. During the initialization phase, KzicIO configures soft timers in addition to the activities explained in Section 3.3.1.
(2) ***Monitoring memSB and prepare I/O requests***. Every trigger state (e.g., interrupt handlers) of soft timers invokes the control logic of KzicIO, monitoring memSB (❶). Based on metric information, KzicIO prepares I/O requests with scheduled release times and inserts them into our timer wheel structure (❷).
(3) ***On-time issuing of the scheduled I/O requests***. At each trigger event, KzicIO peeks the timer wheel to determine which I/O requests are due for issuing. Most I/O requests are issued within interrupt handlers (❸), ensuring data reaches the data buffer just in time (❹). But, if the buffer reaches full capacity, a soft IRQ is scheduled to re-trigger the I/O scheduling process. This ensures that I/O requests are issued at a point when the DBMS workers will have consumed enough data from the buffer.
(4) ***Updating memSB upon data arrival***. Once a storage device completes the DMA transfer, it raises an interrupt (❺). This interrupt triggers the soft timers, which update the buffer's status in memSB to '`ready`', informing UzicIO that the data buffer is now available for the DBMS workers to access (❻).

3.4.3 ***Rate-adaptive scheduling of I/O requests***. The insight for scheduling I/O requests is to *align the data prefetching rate with the DBMS ingestion speed* and to compute the future release times based on the statistics maintained in memSB. The basic principle is to ensure that data arrives just before it is needed by scheduling subsequent requests with continually adjusted future release times. The key control parameters and equations are outlined as follows:

- ***Device throughput and DBMS ingestion speed***. KzicIO measures the device throughput $S_d(t)$ for every I/O request [52] while UzicIO calculates DBMS ingestion speed $S_u(t)$ at time t for each zicIO channel. The flow control logic should satisfy the following equations in stable conditions (i.e., $S_d(t) \geq S_u(t)$), where we have a request with the size of N bytes, target interval Δ_u derived from $S_u(t)$, and estimated interval and release delay: Δ_d and Δ_r, respectively.

$$\Delta_u = \frac{N}{S_u(t)}\,, \qquad\qquad \Delta_d = \frac{N}{S_d(t)} + \Delta_r$$

- ***Rate-adaptive I/O flow control***. Assuming $\Delta_u = \Delta_d$ in stable conditions, we derive the future release time T_r based on the current time T_c as follows:

$$T_r = T_c + \Delta_r, \text{ where } \Delta_r = N \cdot \frac{S_d(t) - S_u(t)}{S_d(t) \cdot S_u(t)} \tag{1}$$

  Equation 1 holds as long as $S_u(t) \leq S_d(t)$, and if $S_u(t) = S_d(t)$ (i.e., $\Delta_r = 0$), we release the request immediately. If a single I/O request is insufficient for the DBMS's ingestion speed, additional reward requests (R) are added until $S_u(t) \leq R \cdot S_d(t)$. Hence, KzicIO keeps monitoring $S_u(t)$ and $S_d(t)$ to adjust T_r for *rate-adaptive release time estimation* and uses T_r for I/O requests. Equation 1 enables on-time data delivery for rapid data ingestion.

4 CO-DESIGN FOR OS-LEVEL PAGE SHARING

This section presents SKzicIO, our DB-OS co-design for OS-level page sharing to reduce I/O redundancy. SKzicIO is *sharing-enabled* KzicIO that employs a sliding-window cache and dynamic remapping of OS pages to the database's page table. Hence, SKzicIO relies heavily on page table manipulation.

4.1 Overview of SKzicIO Designs

The core of SKzicIO is a shared page pool (i.e., a sliding window cache) with page table manipulation for sharing OS pages.

4.1.1 ***Sliding-window caching for page sharing***. SKzicIO utilizes a shared pool of OS pages to hold an incoming stream of file chunks, conceptually representing a sliding window of shareable chunks. These OS pages are shared among DBMS workers, employing proper reference counting [9] to ensure safe recycling. The pages become visible to the workers once SKzicIO maps them to the worker's page table and updates the status bits in memSB. Workers then read data pages from the *current* point of the already loaded file streams within this *sliding stream window*, ingesting data at their own pace. After the DBMS consumes the file data, SKzicIO recycles the corresponding OS pages by unmapping them from the page table. Consequently, SKzicIO facilitates reduced I/O redundancy for DBMS workers ingesting data from the same table.

4.1.2 ***Page table manipulation (mapping and unmapping)***. One of the central components of SKzicIO is its mechanism for safe page table manipulation during OS-level page sharing. To prevent serious failures, we maintain a *safety condition* ensuring that UzicIO never inadvertently returns unmapped buffers to DBMS workers, and that SKzicIO never maps new OS pages into data buffers already in active use. Any deviation from this condition would be disastrous for zicIO. Our solution involves memSB, which uses status bits to arbitrate concurrent access between UzicIO and SKzicIO. The following status transition rules detail how each component ensures safe manipulation of memSB entries:

- ***Status transition by UzicIO***. UzicIO changes the state of memSB from '`in-use`' to '`done`' as expected. But, it must use a CAS operation to change the state from '`ready`' to '`in-use`' since SKzicIO may *forcefully* unmap unused '`ready`' OS pages from the worker's page table. This precaution is necessary if the database worker ingests data too slowly (i.e., outlier), leading to rapid depletion of available OS pages in the page pool.
- ***Status transition by SKzicIO***. SKzicIO, upon detecting the memSB entry marked as '`done`' with its *reference counter* being zero, changes the state to '`empty`', and later changes the state of memSB to '`ready`' only after it *pre-maps a new shareable OS page to the page table* of the DBMS worker and flushes obsolete TLB entries. The forceful unmapping requiring a CAS operation arises to handle outliers (Section 4.2.3).

This coordination ensures that OS pages are safely and transparently managed, allowing DBMS workers to access data without direct involvement in the page-sharing operations.

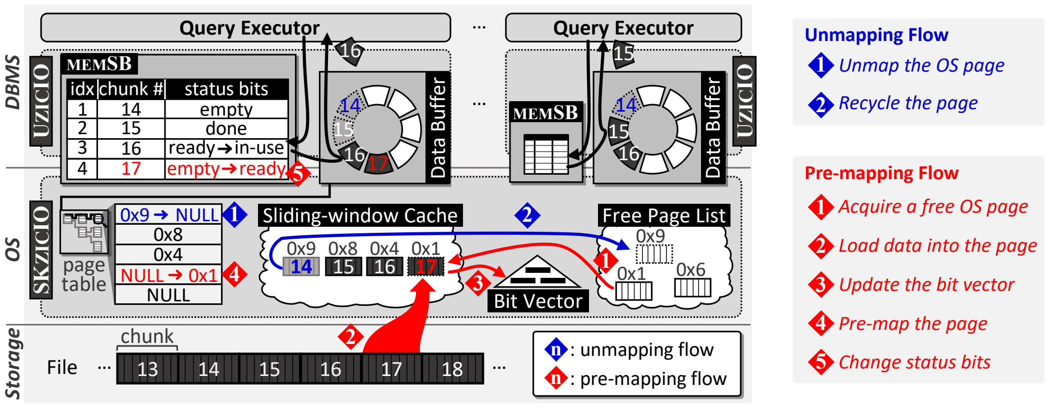


Fig. 4. Page table manipulation for OS-level page sharing.

4.2 OS-level Page-sharing Architecture

We first explain new data structures and basic sharing logic, including methods—i.e., *pre-mapping* and *forceful unmapping* of OS pages—to handle DBMS workers with various ingestion speeds. Then, we describe our flow control logic for page sharing.

4.2.1 ***Data structures for page sharing***. Unlike KzicIO, where OS pages are pinned for data buffers, SKzicIO manages OS pages as a shared pool with associated search data structures and dynamically maps shareable OS pages to the page table of a DBMS worker, i.e., data buffers. SKzicIO has the following data structures:

- ***Sliding-window cache***. An OS page pool is initialized with a collection of empty OS pages, designed to act as a sliding window cache for storing 2 MiB-sized file chunks. Since a logical database table may span multiple file segments, we create a single shared pool for a group of logically related files. Additionally, each shared pool permits a single zicIO channel to efficiently fetch data during page sharing (Section 4.2.4).
- ***Bit vector and chunk hash table***. Each shared pool is equipped with a bit vector and a chunk hash table to assist SKzicIO channels in identifying the current head of the stream window and locating shareable file chunks. The bit vector represents the logical positions of 2 MiB-sized chunks within shared file streams, indicating whether specific file chunks exist in the shared pool. For concurrency, the chunk hash table is protected using Linux `rcu` locks [53].
- ***Reserved page table entries for stream sharing***. SKzicIO pre-allocates a 1 GiB virtual address space, corresponding to a 4 KiB *page middle directory* (`pmd`) table that covers 512 huge pages. It then marks the available memory within this space as '`empty`' in MemSB. Since DBMS workers are unaware of the presence of unmapped address space, we refer to this as *bogus* address space. These inactive entries transition to active status once SKzicIO maps shareable pages.

4.2.2 ***The OS page life cycle in a shared pool***. SKzicIO recycles OS pages within a shared pool as it processes streams of file chunks, making efficient page recycling essential for effective resource management. To provide insight into this process, we outline the OS page life cycle from the perspective of page sharing (**labels refer to Figure 4**):

(1) ***Available for sharing***. SKzicIO begins by obtaining a free OS page from the free page list and identifying the next file chunk to read (1). After the chunk is loaded into the OS page (2), the page is made shareable in the pool through the chunk hash, assigned with a *future expiration time*. The bit vector is then updated to mark the page as available for sharing in the pool (3).
(2) ***Shared through pre-mapping***. For a DBMS worker, SKzicIO identifies a shareable OS page in the pool using the bit vector and verifies its validity through the chunk hash. Upon validation,

the page is shared by incrementing its reference counter, *pre-mapping* it into the DBMS worker's page table (❹), and updating its status bits in memSB to '`ready`' (❺). The DBMS worker can then ingest the data at its own pace.

(3) ***Unmapped and recycled after expired***. When an expiration timeout occurs, SKzicIO updates the status in memSB to '`empty`', unmaps the OS page from the DBMS worker's page table (❶), and decrements its reference counter. If the counter reaches zero, the page is eligible for eviction from the window cache and can be returned to the free page list for recycling (❷).

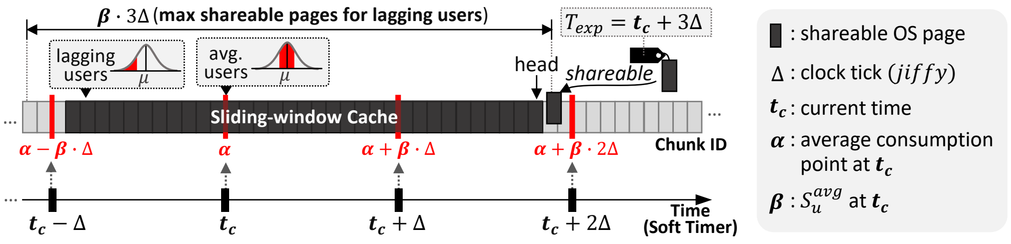
4.2.3 ***OS-level page-sharing logic at scale***. DBMSs must initialize a shared pool for related files associated with a specific DBMS table. Upon creating the shared pool, SKzicIO assigns a unique access key to the newly established pool. A DBMS worker then uses this access key to create a zicIO channel, enabling OS-level page sharing through the shared page pool. Unlike KzicIO, most SKzicIO channels primarily pre-map or unmap shareable OS pages to or from the DBMS worker's page table, rather than issuing I/O requests. For efficiency, only one channel manages I/O control. These SKzicIO channels execute the sequence (↪❹→❺⟶❶→❷↩) illustrated in **Figure 4**. However, if DBMS workers ingest data at a slower pace, SKzicIO channels may pre-map only a limited number of pages, causing numerous skipped chunks that must be reloaded from storage, leading to redundant fetch operations. To mitigate this issue, we enforce a policy requiring all workers to *derail* from the shared pool after a *single round*, allowing them to independently retrieve any missing data. Once derailed, SKzicIO employs local bit vectors to identify and reload the skipped file chunks. The following general rules are applied to handle outliers effectively.

General rules for handling outliers in page sharing. In our OS-level page-sharing environments, workers may exhibit significant variations in data ingestion speeds, with some processing data much faster or slower than others. To address these outliers effectively, our policy prioritizes the majority of workers operating at similar speeds while managing the following cases:

- ***Preventing retention efforts***. DBMS workers are restricted from simultaneously retrieving multiple buffers from our library, ensuring that resources are not intentionally retained.
- ***Limiting the pool capacity to ensure pace***. SKzicIO enforces a limit on pool capacity to ensure that all workers operate at a similar pace. This prevents faster workers from advancing too far ahead, maintaining balance and fairness within the shared environment.
- ***Forceful unmapping***. When SKzicIO detects `ready`' but *expired* pages mapped to DBMS page tables, it *forcefully unmaps* these pages after atomically updating their state to `empty`' in memSB using a compare-and-swap (CAS) operation. Since UzicIO also uses CAS to transition a page's state from `ready`' to `in-use`', the page is either safely unmapped by SKzicIO or properly utilized by UzicIO. This policy ensures that a single unresponsive DBMS worker can hold at most one page, while SKzicIO recycles the remaining *ready* but *unused* pages for other workers, maintaining efficient resource utilization.

4.2.4 ***Adapting I/O flow control for page sharing***. Our OS-level page-sharing logic resembles the traditional DBMS approach for *scan sharing* [25, 42, 57], where newly joined scans synchronize with existing scan sharers. To maintain fairness and efficiency, we refine our flow control logic to prioritize the majority of DBMS workers over individual outliers. Aligned with the rationale in Section 3.4.3 regarding I/O scheduling, SKzicIO aligns the I/O rate with the average ingestion speed, $S_u^{avg}(t)$. Specifically, the refined flow control logic mandates workers within the same sliding-window cache to replace their individual ingestion speed $S_u(t)$ with $S_u^{avg}(t)$ in Equation 1.

The core efficiency insight lies in restricting the initiation of I/O requests to a single SKzicIO channel when needed. *Simultaneous casting by multiple channels requires many pages, rapidly depleting available OS pages and causing control instability.* As a result, only one SKzicIO channel handles I/O requests (i.e., repeating this sequence (↪❶→❷→❸→❹→❺⟶❶→❷↩) shown in **Figure 4**). The remaining channels concentrate on page-sharing operations.



$\beta \cdot 3\Delta$ (max shareable pages for lagging users)
$T_{exp} = t_c + 3\Delta$
lagging users μ
avg. users μ
head
shareable
Sliding-window Cache
$\alpha - \beta \cdot \Delta$
α
$\alpha + \beta \cdot \Delta$
$\alpha + \beta \cdot 2\Delta$
Chunk ID
$t_c - \Delta$
t_c
$t_c + \Delta$
$t_c + 2\Delta$
Time (Soft Timer)
▮ : shareable OS page
Δ : clock tick $(jiffy)$
t_c : current time
α : average consumption point at t_c
β : S_u^{avg} at t_c

Fig. 5. Adapting I/O flow control for page sharing.

Slow soft timers for most sharers. As the refined flow control logic requires one channel to handle I/O operations, the remaining channels do not receive I/O interrupts and thus require an alternative timer to continue pre-mapping shareable OS pages for their respective DBMS workers. To achieve this, we employ a timer event registration with Linux local APIC timers, which awakens the `softirq` daemon at a slow clock pace (i.e., Linux '`jiffies`' ~ 4 ms). Hence, the `softirq` daemon activates the flow control logic (i.e., *pre-mapping*) every 4 ms for most workers, while interrupt-driven soft timers keep a single channel responsible for I/O handling.

Prefetching data for sharing. The main rationale of SKzicIO is to prefetch data that DBMS workers can consume within a single long clock tick (i.e., 4 ms) operating under slow soft timers. Therefore, the high watermark of the sliding window should be set at a maximum of two clock ticks (=2·`jiffies`) to ensure that enough data is prepared for any nimble worker to consume data without waiting. As depicted in Figure 5, this insight leads to the target range for controlling the *head* of the stream window as follows: $\alpha + \beta \cdot \Delta \leq head \leq \alpha + 2 \cdot \beta \cdot \Delta$, where Δ denotes Linux '`jiffies`' with t_c, α, and β being explained in the figure. Automated flow control regulates I/O requests to keep the head of the stream window within this range, allowing even lagging workers to ingest shareable data pages safely at a slow clock pace.

Adjusting expiration timeout (T_{exp}). Upon the shared page pool reaching its high watermark with data pages, it becomes necessary to establish an expiration timeout for the timely recycling of shared pages. As shown in Figure 5, lagging workers will not fall behind by more than one Δ interval from the average consumption point (i.e., $\alpha - \beta \cdot \Delta$), as the *forceful unmapping* policy promptly unmaps unused OS pages from the page table. Hence, a newly prefetched page positioned at the *head* of the stream window (i.e., $\alpha + \beta \cdot 2\Delta$) must be alive until lagging workers can safely ingest it. To this end, we set T_{exp} to $t_c + 3\Delta$ for the newly loaded page, where t_c is the current time. Consequently, the maximum capacity of the shared page pool is $\beta \cdot 3\Delta$ (see Figure 5). Another scenario arises when workers are unable to ingest a single 2 MiB page within one Δ interval. In such cases, our logic prepares only one more page and sets the timeout to a sufficient duration (i.e., 2·huge pages$/S_u^{avg}(t)$) to allow any worker to consume the pages.

5 Evaluation

This section presents the evaluation results of three database engines with and without zicIO under standard data warehouse workloads supplemented by microbenchmarks to validate our claims. Our evaluation focuses on the following three questions:

(1) ***Any regression?*** We measure CPU cycles required to process warehouse queries to identify any hidden costs associated with using the zicIO interface in existing database engines. For an in-depth analysis, we use a data ingestion microbenchmark using `io_uring` and `pread` system calls and measure performance metrics with various configurations.

(2) ***Does zicIO truly liberate DBMSs from data access control?*** We evaluate the overall data access latency of data warehouse queries and analyze the latency breakdown to verify the

Table 1. UzɪᴄIO (`libzicio`) APIs and descriptions

API Name	Description	DBMS	Compatibility
`zicio_open()`	Create a zɪᴄIO channel	Postgres, MySQL (MyISAM), Citus, DuckDB	KzɪᴄIO, SKzɪᴄIO
`zicio_close()`	Close the zɪᴄIO channel	Postgres, MySQL (MyISAM), Citus, DuckDB	KzɪᴄIO, SKzɪᴄIO
`zicio_get_page()`	Get the next page from UzɪᴄIO	Postgres, MySQL (MyISAM), Citus, DuckDB	KzɪᴄIO, SKzɪᴄIO
`zicio_put_page()`	Return the page to UzɪᴄIO	Postgres, MySQL (MyISAM), Citus, DuckDB	KzɪᴄIO, SKzɪᴄIO
`zicio_create_pool()`	Create a sliding-window cache	Postgres, MySQL (MyISAM)	SKzɪᴄIO
`zicio_notify_ranges()`	Provide the ranges to UzɪᴄIO	Citus, DuckDB	KzɪᴄIO

* All APIs are non-blocking. † Rate of change is calculated using `rdtsc` [26].

central claim that our DB-OS co-design for automated I/O control faithfully addresses all five latency factors (**L1**-**L5**).

(3) ***How effective is OS-level page sharing?*** We first confirm the limitation of a simple zero-copy, OS-bypassing approach for multiple scans on the same table. We then conduct comparative studies against DBMS scan sharing and discuss the advantages and disadvantages of our DB-OS co-design approach for addressing I/O redundancy, which may easily arise when attempting to leverage zero-copy OS-bypassing designs.

5.1 Evaluation Setup

5.1.1 ***Server configurations.*** For our evaluation study, we employed a 128-core machine featuring two AMD EPYC 7742 processors, 512 GiB of DRAM, and SK hynix P41 PCIe 4.0 SSD capable of delivering 7 GiB/s for sequential reads. We implemented the OS components KzɪᴄIO and SKzɪᴄIO in Linux-5.15 and developed UzɪᴄIO as a user library, `libzicio`, with six core APIs integrated into DBMS codebases (see Table 1).

5.1.2 ***Database adaptation.*** Our evaluation uses four disk-based, open-source databases: MySQL-8.0 with the MyISAM storage engine, PostgreSQL-15.0 with the default row-wise heap storage engine, PostgreSQL with the Citus [10] columnar engine, and DuckDB-1.1 [11, 47] with the Parquet [3] columnar format. Each supports full SQL semantics but differs in its ability to generate optimal query plans. We regard the DBMS buffer manager as the main entry for data access and inject UzɪᴄIO hooks to invoke our APIs instead of legacy buffer manager APIs only when the query plans specify *sequential or range scans* as the table access method. Consequently, zɪᴄIO plays a one-way bridging role from storage to DBMS workers, and we pass buffer pointers to query execution layers to ensure that zɪᴄIO remains user-friendly for legacy databases. For non-sequential data access scenarios, such as index lookups, the legacy buffer pool is still utilized to maintain backward compatibility.

Modifications to columnar DBMSs. DBMSs that rely on columnar formats [2, 3, 10]—such as Citus and DuckDB—use hierarchical storage layouts aligned with the PAX model [1]. This model organizes data into a set of record groups (e.g., *stripes* in Citus), with each group containing multiple column chunks. These record groups are stored contiguously across multiple files (e.g., DuckDB's Parquet files). The PAX-like layout is prevalent in many open-source columnar formats, ensuring that our adaptation strategy generalizes to other columnar engines. A key precondition for our approach is that these formats need to allow zɪᴄIO to know the location of required data beforehand to facilitate efficient scanning on columnar storage through multiple range scans. The necessary range information is retrieved by reading metadata—specifically, the Postgres catalog in Citus and `FileMetaData` in DuckDB's Parquet format—that contains the location information for column chunks within each record group. To this end, we use Citus' `ColumnarStorageRead()`, whereas we utilize DuckDB's `ParquetReader::ScanInternal()` function. Once the necessary range information is collected, it is passed to UzɪᴄIO via `zicio_notify_ranges()`.

5.1.3 ***Database tuning knobs.*** We configure the '`work_mem`' parameter of PostgreSQL to 128 MiB for efficient `join` processing. Additionally, we enable LLVM-based JIT query compilation [55, 58] to accelerate simple operations and expedite our evaluation. While other performance tuning options, such as parallel scanning and data partitioning, are worth considering, we chose not to tune these options, as database tuning is beyond the scope of our work. Lastly, for profiling purposes, we modified the databases to measure query latency, per query I/O delay, and per query zicIO metrics (e.g., waiting time when overloaded).

5.1.4 ***Data warehouse workloads.*** We use the standard TPC-H benchmark to emulate data warehouse workloads with a scale factor of 100. We use HammerDB-4.6 [18], which faithfully follows the TPC-H specifications for TPC-H experiments. HammerDB populates 119 GiB data in MyISAM, 170 GiB data in PostgreSQL (heap), 179 GiB data in PostgreSQL (Citus), and 93 GiB data in DuckDB, including 30 GiB, 21 GiB, and 23 GiB index files for MyISAM, PostgreSQL (heap), and PostgreSQL (Citus), respectively. Note that no index is used in DuckDB (Parquet) [12], and three tables—`LINEITEM`, `ORDERS`, and `PARTSUPPLY`—occupy the most space. However, users create shared page pools for any eligible tables and let database workers join one of the pools on demand.

5.1.5 ***Overall development effort.*** In developing zicIO, we made extensive modifications to integrate it seamlessly into the Linux kernel and database engines. However, the changes to Linux were relatively modest, spanning 1,802 source lines of code (SLOC), compared to the codebase required for zicIO itself: 17,882 SLOC for KzicIO and SKzicIO, and 1,122 SLOC for UzicIO. Additionally, we made code modifications in the database engines; (**i**) 1,313 SLOC to integrate zicIO into MyISAM for MySQL; (**ii**) 765 SLOC for Postgres (heap); (**iii**) 736 SLOC (480 SLOC for Citus and 256 SLOC for Postgres) for Postgres (Citus); (**iv**) 497 SLOC for DuckDB (Parquet). The detailed breakdown of SLOC for zicIO components and the modifications required for DBMS adaptation is provided in Table 2.

Table 2. The breakdown of source lines of code (SLOC)

Source lines of code in KzicIO & SKzicIO (OS module)

linux/zicio/*.*	SLOC	linux/zicio/*.*	SLOC	linux/zicio/*.*	SLOC
zicio.{c|h}	1462	zicio_desc.{c|h}	632	zicio_atomic.h	57
zicio_mem.{c|h}	1725	zicio_extent.{c|h}	915	zicio_ghost.{c|h}	679
zicio_files.{c|h}	571	zicio_flow_ctrl.{c|h}	794	zicio_shared_pool.{c|h}	4931
zicio_cmd.{c|h}	2205	zicio_req_timer.{c|h}	1051	zicio_shared_pool_mgr.{c|h}	226
zicio_device.{c|h}	672	zicio_req_submit.{c|h}	566	zicio_data_buffer_descriptor.{c|h}	218
zicio_notify.h	96	zicio_md_flow_ctrl.{c|h}	934	uapi/zicio.h	148

Source lines of code in UzicIO (user library)

libzicio/*.*	SLOC	libzicio/*.*	SLOC
zicio_lib.c	992	libzicio.h	130

Source lines of code in Linux-5.15 modifications

Modified files in linux	SLOC
pgtable_types.h, pci.c, extents.c, main.c, fops.c, tlb.c, direct-io.c, read_write.c, blk-mq.h, blk_types.h, iov_iter.c, sched.h, syscalls.h, blk-mq.c, exit.c, fork.c, memory.c, mmap.c, core.c, idle.c, fs.h, etc	1802

Source lines of code in DBMS modifications

DBMS engine	Modified files in each engine (only substantially changed ones are listed)	SLOC
MySQL (MyISAM)	mi_scan.cc, mi_dynrec.cc, mi_stat_rec.cc, mi_write.cc, myisamdef.h, etc	1313
Postgres (heap)	heapam.c, etc	765
Postgres (Citus)	columnar_reader.c, columnar_storage.{c|h}, etc	736
DuckDB (Parquet)	parquet_reader.cpp, thrift_tools.hpp, etc	497

5.2 Data Ingestion Microbenchmark

5.2.1 ***Experimental setup.*** We evaluate the performance metrics of our microbenchmark that directly invokes UzicIO APIs, `pread()`, and `io_uring` system calls to emulate full and range scans. The microbenchmark is configured to explore how `io_uring` reduces known I/O latencies (**L3**-**L5**) using three key features: (1) `O_DIRECT` to bypass the OS page cache layer, (2) dedicated `io_uring` threads for submission queue polling [48], and (3) prefetching data to mitigate the limitations of blocking I/O. In contrast, the benchmark that utilizes `pread()` employs two configurations: (i)

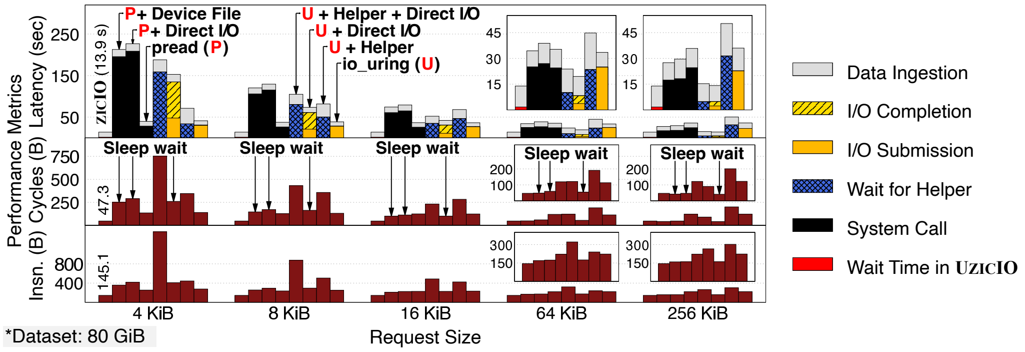


Fig. 6. Performance metrics under sequential ingestion.

a regular file and (ii) a *device file* (e.g., open("/dev/nvme0n1",..)) that circumvents file system stacks and replaces the *deprecated raw device*.

Performance is evaluated across varying request sizes (4 KiB to 256 KiB), with all measurements taken after flushing the OS page cache to simulate a cold cache environment. The latency metric breaks down different phases of the I/O process. For example, a single label ("*Wait for Helper*") represents the time the microbenchmark thread waits for the dedicated io_uring helper thread to complete data delivery. This aggregated latency can be further decomposed into "*I/O Submission*" and "*I/O Completion*" phases if the microbenchmark handles I/O operations directly. These breakdowns are illustrated in Figure 6 and Figure 7. **UzɪcIO is paired with KzɪcIO for this evaluation**.

5.2.2 ***Sequential data ingestion***. To emulate a full scan, we sequentially read 80 GiB of data. We measured performance metrics for io_uring, pread() w/ and w/o a *device file* by varying data sizes. However, zɪcIO treats the entire data as a single batch, so the request size is dynamically adjusted by our microbenchmark. Figure 6 shows the performance metrics of various I/O schemes. Data ingestion using the blocking I/O interface—pread() w/ and w/o a *device file*—generally exhibits more I/O wait time than zɪcIO and io_uring with large data (> 64 KiB). This increased wait time is due to the mode switch that occurs when the pread() system call submits I/O requests, processes them through the OS cache layer (**L4**), and deals with the physical I/O latency that cannot be mitigated by the blocking I/O mechanism.

Direct I/O has limited impact on sequential scans. Interestingly, the benefits of direct I/O diminish as request sizes decrease. Although pread() with a *device file* outperforms pread() with O_DIRECT by bypassing storage layers such as the file system, it still lags behind OS-cached access for smaller requests. In fact, Linux's default prefetching mitigates the negative impact of storage stack latency, leading to better performance than direct I/O (O_DIRECT and *device file*) across all cases. Notably, the default I/O mechanism—pread without O_DIRECT, widely used in legacy DBMS engines—demonstrates consistent performance across data sizes and even surpasses all direct I/O configurations for smaller data. This finding highlights that while direct I/O can enhance performance for specific workloads, it is not always the optimal choice for varying data sizes. Thus, a subtle policy is necessary when deciding whether to employ direct I/O.

Data prefetching obviates the need for helper threads. Another interesting observation is that data ingestion with the optimized io_uring interface exhibits a widening performance gap with pread as the data size decreases. Although io_uring aims to hide I/O latency through asynchronous I/O by dedicating threads to submission queue polling, it spends significant cycles waiting for the OS threads to deliver requested data ("*Wait for Helper*") due to the highly frequent submission of I/O requests. Figure 6 shows that with 4 KiB requests, this waiting activity, backed by high CPU cycles ('Cycles') and instruction counts ('Insn.'), is notable compared to methods without helpers that

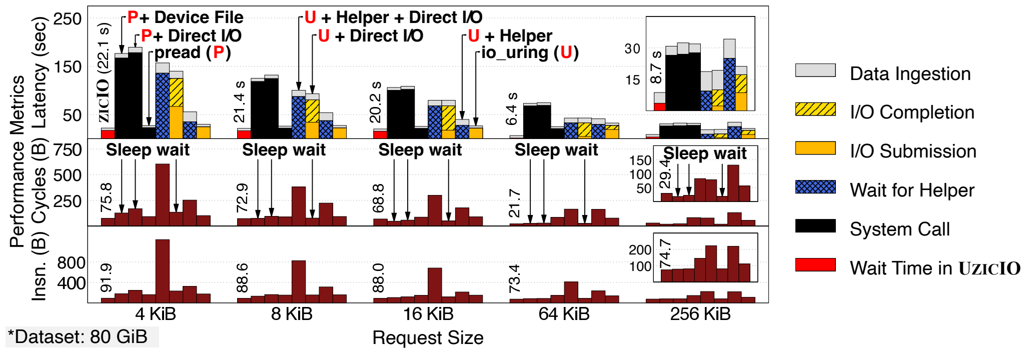


Fig. 7. Performance metrics under disjoint range scans.

sleep-wait ("*Sleep wait*") for I/O completion or read data already prefetched by the Linux storage stack. By contrast, zicIO keeps I/O wait times to a minimum in these microbenchmark scenarios because zicIO can prefetch data without involving the host worker. This carefully orchestrated approach ensures timely data delivery by decoupling the data access mechanism from the host worker, effectively addressing all five of the identified latency factors.

5.2.3 ***Data ingestion with disjoint range scans.*** To simulate data ingestion using disjoint range scans, we split the 80 GiB dataset into batches of different sizes and scanned only the batches with odd numbers, resulting in a total of 40 GiB being scanned. When using small-sized batches, data ingestion involves many small I/O requests internally, representing the most challenging condition for any I/O interface, including zicIO. For a fair comparison, we configured `io_uring` to issue I/O requests for the next data batch in advance to promote data prefetching through its asynchronous I/O (i.e., OS helper threads). Our evaluation reveals interesting phenomena, e.g., the versatility of the default `pread()`. Although further investigation is needed, we found two notable observations:

(1) ***Small-sized range scans.*** The effectiveness of the default `pread()` with the Linux storage stack in handling small-sized batches exceeds our expectations. As shown in Figure 7, small batch sizes place significant pressure on zicIO due to the increased volume of I/O requests processed by KzicIO. Surprisingly, `pread()` with OS prefetching performs exceptionally well for small, disjoint range scans, with only a minor latency difference compared to zicIO, while outperforming all `io_uring`-based ingestion methods. In contrast, direct I/O remains ineffective across all configurations, despite bypassing storage stacks through the *device file* interface and leveraging asynchronous I/O via `io_uring`. This demonstrates that even with optimizations, direct I/O struggles with workloads involving small, scattered requests.

(2) ***Large-sized range scans.*** The asynchronous I/O through helper threads in the `io_uring` interface is less effective in large-sized range scans than sequential ones. With large batches, zicIO streamlines data prefetching using the provided batch information, whereas `io_uring` methods, despite having prior knowledge of data access patterns, spend significant CPU cycles on "*Wait for Helper*". Thus, the performance metrics for `io_uring` in multi-range scans are worse than in sequential scans, and the performance gap with zicIO widens. Our evaluation indicates that non-sequential reads, like our multi-range scans, impact all I/O methods badly, underscoring the effectiveness of the automated data access control of zicIO.

5.3 Evaluation with Legacy DBMSs

5.3.1 ***Experimental setup.*** We evaluate the performance metrics of simple data analytics using TPC-H queries (see all SQL queries (Q1–Q22) [19, 20] defined in HammerDB-4.6 that we used for MySQL (MyISAM), Postgres (heap), Postgres (Citus), and DuckDB (Parquet) that may or may not perform sequential scans, depending on generated query plans. To emulate data access in a cold or

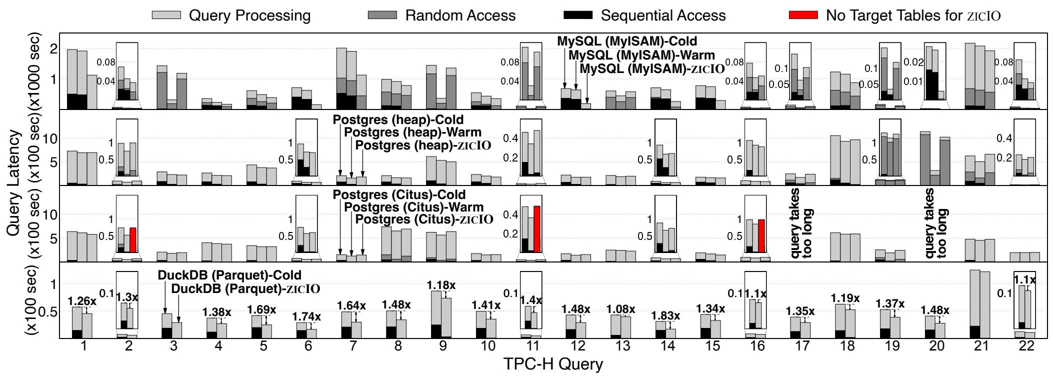


Fig. 8. Latency of TPC-H queries on four database engines with and without zicIO.

warm cache, we configure a single database client to send all TPC-H queries with the OS page cache flushed before starting the next one. For each query, we run it twice to observe the differences between cold and warm cache environments. Unless stated otherwise, we enable Linux *readahead* for all experiments.

We decompose query latency into three key components: (i) *query processing time* in memory, (ii) *random data access* (e.g., index lookups), and (iii) *sequential data access* (e.g., scans). For precise measurement, we capture data access latency around the `pread()` system call in MySQL (MyISAM), as MyISAM lacks a dedicated DBMS buffer layer. In Postgres variants, we measure latency through buffer manager APIs: `ReadBufferExtended()` for Postgres (heap) and `ReadBuffer()` for Postgres (Citus). In DuckDB, we monitor data access latency around the `pread()` call because, when working with Parquet files, DuckDB (Parquet) follows distinct call paths that bypass its default buffer pool, thus operating without its internal buffer cache (refer to this discussion [13] for more details). As a result, DuckDB (Parquet) performs direct I/O operations, impacting data access performance differently compared to its typical buffer-managed workflows. Overall, we provide a granular breakdown of the time spent at various stages of data access and query processing, offering deeper insights into the performance characteristics of each database engine across different workloads. **As in previous evaluations, UzicIO is integrated with KzicIO in this set up**.

5.3.2 ***Evaluation with MySQL (MyISAM)***. For this experiment, we use three configurations for MySQL: cold and warm data access in MySQL on vanilla Linux ('*MySQL-Cold*' and '*MySQL-Warm*') and MySQL with zicIO ('*MySQL*-zicIO'). We configure MyISAM to have a large key cache (30 GiB) to cover MyISAM index files. Figure 8 shows the latency breakdown for TPC-H queries running on MySQL. In all experiments, data access latency takes a significant time in many queries involving the giant `LINEITEM` table. In contrast, MySQL-zicIO eliminates all sequential I/O overhead and sometimes outpaces MySQL-Warm. MySQL-zicIO takes much shorter than *MySQL-Cold* for queries (e.g., Q20 (6.13×)). As expected, analytic queries whose query plans have index access methods—Q3, Q9, Q11, Q13, and Q19—benefit less from zicIO than those performing sequential scans. From now on, we will use a group of queries having sequential scans on the `LINEITEM` table and use the query set in the following experiments for MySQL and PostgreSQL.

5.3.3 ***Evaluation with PostgreSQL***. For evaluation with PostgreSQL, we use three configurations similar to those for MySQL. We configure PostgreSQL to have a big buffer cache (220 GiB) and run the same test twice, which gives two behaviors: cold access to storage ('*Postgres-Cold*') and warm data access in the cache '*Postgres-Warm*'). In the warm buffer cache, query processing in PostgreSQL should be completed without accessing storage, but it spends time in the buffer manager to spot target database pages, which is counted as the DBMS buffer overhead. In Figure 8, query latencies and their internal latency breakdown look slightly different from MySQL, mainly due to the difference in query plans and LLVM-based query compilation in Postgres. Another noticeable result is the nontrivial I/O overhead in Postgres with a 220 GiB warm cache. An in-depth analysis of its codebase answers our inquiry; Postgres internally uses ring buffers if the scanned table occupies a quarter of the total buffer capacity [56]. zicIO efficiently helps Postgres perform rapid table scanning in such queries by preparing database pages right before needed. With a single client configuration and large memory, vanilla Postgres exhibits good performance, so Postgres-zicIO slightly enhances performance in Q6-Q9.

5.3.4 ***Evaluation with columnar engines.*** For this experiment, we used three configurations for Citus: cold and warm access in standard Citus (referred to as '*Citus-Cold*' and '*Citus-Warm*'), and Citus with zicIO (denoted as '*Citus-zicIO*'). In contrast, we employed only two configurations for DuckDB—cold access in standard DuckDB ('*DuckDB-Cold*') and DuckDB with zicIO ('*DuckDB-zicIO*'). This difference is because DuckDB (Parquet) bypasses its buffer pool when accessing Parquet files [13], relying entirely on direct I/O and sequential access without index support [12]. We also disabled parallel processing and data compression in this evaluation. Despite these limitations, DuckDB's query performance consistently outclasses all other engines. Additionaly, zicIO further enhances DuckDB's performance by effectively masking physical I/O delays, thereby minimizing latency during data access (see Figure 8). Citus, even with its PAX-like format, benefited less from zicIO mainly due to its smaller granular range scans compared to DuckDB, which performs larger-sized columnar range scans.

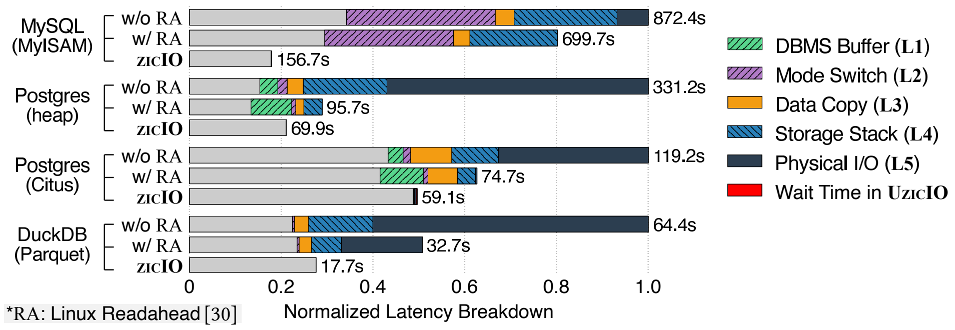


Fig. 9. The normalized latency breakdown for Q6 (cold).

5.3.5 ***Normalized latency breakdown for Q6.*** In our analysis of the normalized latency breakdown of Q6 across all four engines (as shown in Figure 9), we observed that each benefits from Linux readahead to some extent. For MySQL (MyISAM), most of the latency occurs within the Linux storage stack (**L4**) due to the overhead of the OS page cache and frequent mode switches (**L2**). Notably, MySQL effectively hides physical I/O latency (**L5**) by leveraging Linux readahead and prefetching mechanisms. However, it still incurs considerable overhead within the Linux page cache (**L4**). Since the MyISAM engine lacks a buffer pool, there is no additional latency from a DBMS buffer layer (**L1**).

Postgres (heap) with and without Linux readahead exhibits a similar latency breakdown, with an interesting observation: buffer latency (**L1**) is present even when accessing cold data. This same pattern emerges in Postgres (Citus), where the DBMS buffer layer introduces notable delays as it duplicates the required column data into its buffers to optimize future access. In contrast, DuckDB (Parquet) primarily incurs latency from waiting on physical I/O operations (**L5**), followed by storage stack overhead (**L4**). When integrated with zicIO, all four engines improve query performance by reducing data access latency, except in scenarios involving random I/O, such as index lookups or temporary file access.

5.4 Page-sharing Microbenchmark

5.4.1 ***Experimental setup.*** We utilize the same microbenchmark to evaluate the impact of page sharing across various configurations of synthetic workloads. These configurations include differing request sizes (4 KiB–16 KiB: typical page sizes), available memory, and varying numbers of clients simultaneously ingesting the same 80 GiB data file. We set the benchmark ingestion speed to around 1 GiB/s to emulate database query processing. For this evaluation, we compare zicIO with `pread()`-based implementations (i.e., regular or *device files*), as our test databases use `pread` for data retrieval. Our objective is to confirm the significance of redundant data fetching under different configurations and to evaluate how effectively zicIO mitigates this issue through OS-level page-sharing. By adjusting these configuration parameters, we can better understand the efficiency and performance improvements provided by zicIO in reducing unnecessary data fetches and improving data access times. **Now, we will pair UzicIO with KzicIO or SKzicIO**.

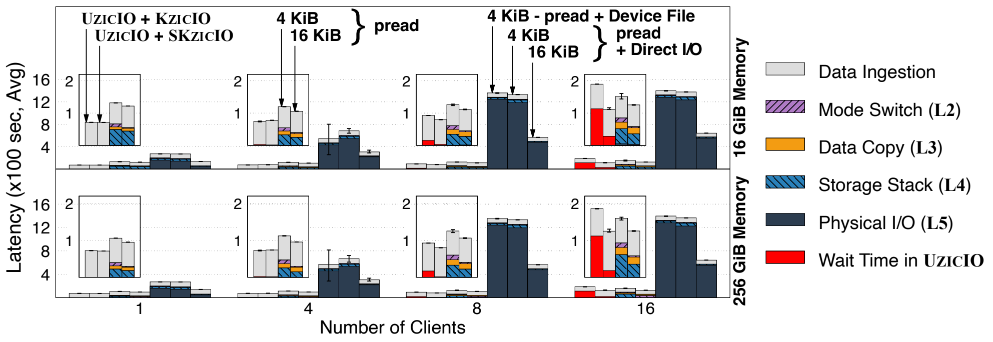


Fig. 10. Microbenchmark with concurrent data ingestion.

5.4.2 ***The more, the better.*** Larger request sizes reduce overhead by minimizing the number of system call invocations. As illustrated in Figure 10, concurrent data ingestion using `pread()` with the OS page cache benefits from fetching larger data chunks, increasing the likelihood of cache hits. Similarly, increasing available memory has a comparable impact on performance. However, zicIO is unaffected by these optimizations due to its automated I/O control mechanism.

5.4.3 ***The less, the better.*** When more clients ingest data simultaneously, all methods experience slowdowns due to the overhead of increased data access. For example, even approaches such as zicIO (KzicIO), which rely on direct or zero-copy I/O, can suffer from redundant data fetching, resulting in higher latency. A practical strategy with existing I/O interfaces involves configuring the DBMS to use large page sizes and deploying it on a system with ample memory and a generous read-ahead setting to enhance the caching effect. zicIO (SKzicIO) alleviates this burden by requiring fewer resources from DBMSs.

Interestingly, as the number of clients increases, the performance of `pread` with a *device file* deteriorates compared to `pread` with `O_DIRECT`. With a single client, `pread` with a *device file* benefits from reduced storage stack overhead (**L4**), performing on par with `pread` with `O_DIRECT` within a margin of error. However, with four clients, performance variance grows due to physical I/O latency fluctuations (**L5**) caused by device contention. When eight clients are active, `pread` with a *device file* performs worse than `pread` with `O_DIRECT`, despite lower storage stack overhead.

5.5 Concurrent Ingestion with Legacy DBMSs

5.5.1 ***Experimental setup.*** Next, we evaluate the performance metrics of databases using TPC-H queries. Our initial focus is on the performance of legacy databases when multiple queries accessing the same fact table run concurrently. In this experiment, we utilize zicIO for test databases without OS-level page sharing. We select a group of queries that previously exhibited substantial sequential

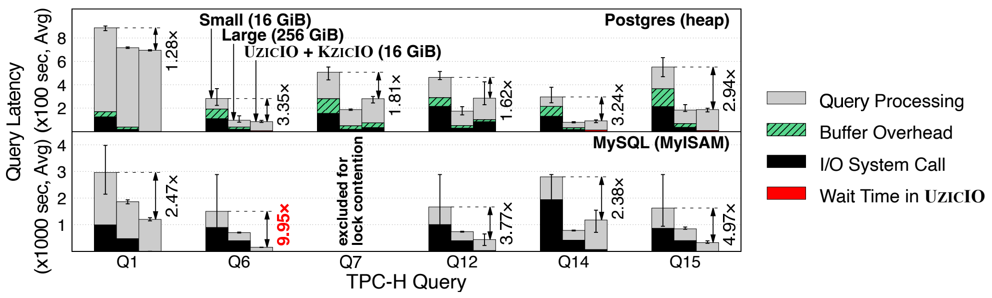


Fig. 11. Query latency with multiple clients.

I/O overhead: Q1, Q6, Q7 for PostgreSQL only, Q12, Q14, and Q15. Multiple clients are created and configured to send all queries randomly, but each query is executed only once from the group, thereby stressing the databases in many ways. The databases are tested under two memory configurations: **small** (16 GiB) and **large** (256 GiB), with ZICIO (KZICIO) set to 16 GiB. Note that the engines with sharing-disabled ZICIO can still streamline data ingestion for all queries unless the device is fully saturated, e.g., Q6 (**9.95**×) (see Figure 11).

5.5.2 ***Evaluation with PostgreSQL.*** In our PostgreSQL tests, we employ seven clients, as this number is sufficient to saturate the device's maximum bandwidth. Figure 11 shows the latency breakdown for TPC-H queries averaged over these seven clients, with minimum and maximum values also indicated. As shown in Figure 11, Postgres-Large exhibits latency behaviors similar to those of a single client in Figure 8 thanks to the large buffer cache. However, when the available memory is reduced, Postgres-Small experiences a significant increase in query latency, particularly in buffer overhead and physical I/O. This is because the time spent performing page lookups and page replacements in a small memory environment results in a substantial impact on all test queries. Worker processes continuously evict pages from the buffer pool, leading to more frequent reads of data pages from storage, thereby incurring higher I/O amounts.

5.5.3 ***Evaluation with MySQL.*** Our MySQL configuration utilizes 20 concurrent clients to fully utilize the maximum throughput of our NVMe device. The latency breakdown for our selected TPC-H queries averaged over 20 clients is shown in Figure 11. MySQL-Large exhibits latency behaviors similar to those in Figure 8, as MySQL worker threads experience negligible resource contention in a large memory setup. Notably, MySQL-Large still incurs nontrivial I/O overhead due to the absence of a DBMS buffer cache. Conversely, MySQL-Small encounters both user-level and OS-level resource contention. Among all queries, Q14 shows a significant increase in I/O overhead, while its query processing time remains relatively stable. Interestingly, MySQL-Small exhibits less overhead than expected. The absence of a database buffer cache seems to provide MySQL (MyISAM) with a degree of resilience to resource scarcity, such as limited memory. Overall, concurrent data ingestion generally increases data access latency when memory pressure is high, and the opportunity to leverage the OS page cache for required data is not fully exploited.

5.6 Page-sharing Evaluation with Legacy DBMSs

5.6.1 ***Experimental setup.*** In the last evaluation, we measured query latency to see the effect of OS-level page sharing as we increased the number of concurrent clients. We modified PostgreSQL and MySQL to pre-create a shared pool for the large `LINEITEM` table to conduct this experiment. We allowed database workers to join one of the shared pools on demand. For PostgreSQL, we enabled "synchronized scan sharing," while the MyISAM storage engine ensured better performance isolation due to the absence of the shared buffer pool. We selected a subset of TPC-H queries, including Q6, Q12, and Q14, for testing page sharing as those queries exhibited the highest ingestion speed (e.g., 1.6 GiB/s for Q6 on PostgreSQL) among all queries, regardless of engine type. We used

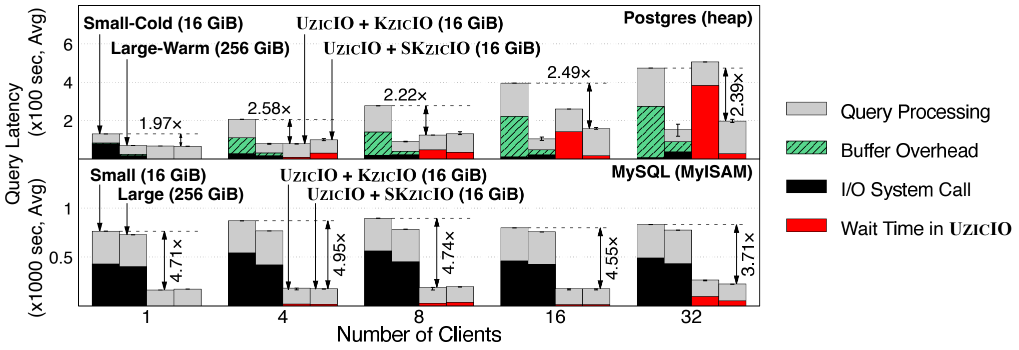


Fig. 12. The effect of OS-level page sharing for Q6.

the same memory configurations and the tuning knob for *synchronized scan sharing*. For all engines, we use zicIO with 16 GiB (see Figure 12).

5.6.2 ***Evaluation with PostgreSQL.*** Vanilla PostgreSQL suffers from high I/O overhead due to its eager activation of ring buffers [56], regardless of memory capacity. As the number of clients increases, vanilla PostgreSQL spends more time on buffer overhead, even with synchronized shared scanning enabled. Expectedly, the engine with a large buffer pool exhibits less overhead in its shared buffer. However, I/O overhead in PostgreSQL with large memory remains noticeable due to the eager activation of the ring buffers. In contrast, zicIO helps Postgres reduce redundant data fetching efficiently. Initially, Postgres-KzicIO experiences increased waiting time in our library (`libzicio`) as the number of clients increases. The growing latency stems from over-subscribed I/O requests, thus saturating device capacity and leading to redundant data fetching. **It confirms the limitation of cache-bypassing and zero-copy data transfer**. However, Postgres-SKzicIO improved the query latency that can be comparable to Postgres-Large (see Figure 12).

5.6.3 ***Evaluation with MySQL.*** As shown in Figure 12, MySQL shows slightly different behaviors compared to PostgreSQL due to the absence of the shared buffer pool. As we have seen in previous results, changing memory configurations would not provide any performance benefit. However, the reduced I/O overhead results in noticeable performance improvements. Compared to PostgreSQL, all vanilla MySQL can sustain query performance, regardless of clients, because the required I/O rate for concurrent users is low. Notably, MySQL-SKzicIO achieved almost matching performance with PostgreSQL, which was an unexpected outcome. **This unforeseen rivalry in performance between the two databases stresses the significance of our automated I/O control with OS-level page sharing**, considering that none of the two database engines, except the storage interface, were altered, nor do we tweak database tuning knobs for favorable results for zicIO.

5.6.4 ***Evaluation with other queries.*** We conducted two more experiments using queries Q12 and Q14 with 32 clients to measure the effectiveness of OS-level page sharing. Like Figure 12, databases can benefit from OS-level page sharing under concurrent query processing. However, PostgreSQL experiences substantial latency in the shared buffer pool with Q12 because its query exploits the index scan to access the `ORDERS` table. For Q14, PostgreSQL exhibits a behavior similar to Q6. On the other hand, MySQL exhibits a severe contention problem. Specifically, there was `mutex` contention in the shared `keycache` when performing the nested loop join processing. We temporarily disabled the key cache to avoid the issue so that MySQL can benefit from SKzicIO. As shown in Figure 13, MySQL-SKzicIO exhibited "*I/O System Call*" with Q12 and Q14 mainly because of unavoidable *random I/O* caused by index lookups and external file sorting for `ORDER BY`.

In-depth analysis through profiling revealed that our soft timers are activated mostly (i.e., 99%) in `softirq` to pre-map shareable pages to user processes or threads when many zicIO channels

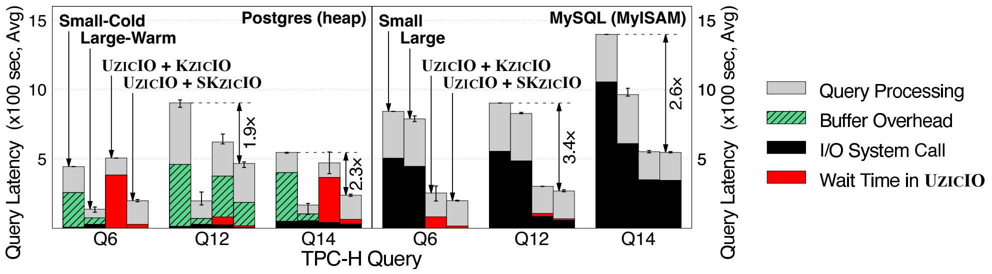


Fig. 13. Query latency with 32 clients scanning LINEITEM.

share the same database table. While supporting OS-level page sharing for 32 concurrent queries scanning the large LINEITEM table, zicIO allocated three huge pages (~8 MiB) for its OS page pool to support MySQL (MyISAM), and it used twelve huge pages (~24 MiB) for PostgreSQL due to the higher ingestion speed. Hence, we validated our claim that **zicIO efficiently mitigates redundant data fetching through OS-level page sharing** for real database systems under analytic queries accessing the same fact tables.

5.6.5 ***Evaluation with other co-design.*** We further explore evaluations with another DB-OS co-design, *vmcache* [36], which aims to reduce buffer cache overhead through virtual memory-assisted, co-designed buffer management. Although its primary focus differs from ours, evaluating its performance across various configurations provides valuable insights for future co-design efforts and opens new research opportunities. For our analysis, we used the *vmcache* codebase [34, 35] along with its buffer manager, adapting our microbenchmark to use their buffer manager APIs to emulate concurrent data ingestion, as shown in Figure 10. We configured the buffer manager with both small (16 GiB) and large (80 GiB) buffer pool sizes to examine performance variations across different memory configurations, similar to the settings in Figure 10.

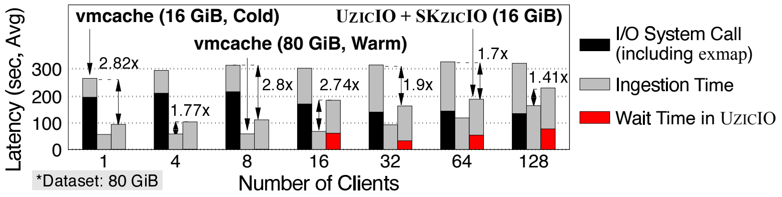


Fig. 14. Different co-designs: vmcache [36] vs. zicIO.

As shown in Figure 14, *vmcache* effectively eliminates the page ID translation overhead in the DBMS buffer through its explicit virtual memory (*exmap* [34]) in environments (i.e., *vmcache* with 80 GiB of buffer memory). However, when memory is constrained, *vmcache* must activate its cache replacement algorithm and retrieve the required data pages from storage devices, resulting in significant data access latency, even for sequential workloads. This pattern persists as the number of clients grows. zicIO also faces growing overhead from reloading skipped chunks under high concurrency, which can lead to device saturation. The evaluation results indicate that a new co-design, which leverages the strengths of both approaches, merits further investigation.

6 Limitations and Future Work

This paper discusses DB-OS co-design to facilitate fast data ingestion for data warehouse systems that handle large static (read-only) datasets. Three limitations require further investigation to broaden its applicability to a wider range of environments.

- ***Mixed workloads.*** zicIO can facilitate data ingestion for analytic workloads operating on large *static (read-only)* datasets. Nevertheless, it has limitations when handling workloads that update

data ingested by applications. The core challenge arises from zicIO's optimized design, which bypasses the Linux file system stack and page cache. Consequently, the current implementation of zicIO must address data consistency—an open problem calling for innovative system designs. One complex workload environment we envision is hybrid transactional and analytical processing (HTAP). As a practical workaround, many commercial solutions employ an ETL (extract-transform-load) approach [21, 32, 33, 38, 41, 49, 60], featuring disaggregated architectures that run dedicated analytic database instances on compressed columnar datasets extracted from separate update-heavy databases. While zicIO can still benefit such analytic databases, HTAP systems inherently involve mixed workloads, and further investigation is needed to determine whether user- or kernel-level designs offer the most viable pathway for ensuring data consistency.
- ***Heterogeneous devices with ultra-low latency.*** Contemporary system designs for large-scale data processing utilize massive parallelism and disaggregated architectures, providing access to remote storage servers over ultra-low latency networks. These environments require data processing systems to ingest streams from diverse devices, presenting significant challenges for both databases and operating systems. Previous studies [16, 23, 29, 39, 44, 46] have proposed solutions to tackle network processing and scheduling issues, focusing on achieving low tail latencies in datacenter applications. However, streamlining data ingestion across heterogeneous devices remains a challenging and open concern for future research.
- ***Various data access patterns.*** Although zicIO significantly streamlines sequential data access and substantially boosts the performance of various analytic queries, database systems continue to rely on traditional data access paths in the DBMS buffer manager for other access patterns, including index-based and random-access methods. Since zicIO was originally designed to optimize sequential I/O, applying its techniques to less predictable or more complex access routines presents new challenges and, in turn, new opportunities. These opportunities extend into research areas that remain largely unexplored, such as specialized graph traversal algorithms and out-of-core matrix computations [31], both of which push the boundaries of conventional in-memory processing. Developing solutions that effectively integrate zicIO's OS-level optimizations with these advanced data manipulation workflows could yield significant performance gains and reveal important insights into next-generation data systems.

7 Conclusion

Rapid data ingestion is crucial for data warehouse systems, and ingesting large fact tables concurrently by database workers presents a significant research opportunity for the database community. To address these challenges, we propose zicIO, a novel DB-OS co-design that liberates DBMSs from the complexities of data access control, enabling them to focus on efficient data ingestion by ensuring that required data is prepared in a timely manner. zicIO encompasses automated I/O flow control and OS-level page sharing, achieved through the development of soft timer-based I/O control automation and a sliding-window cache mechanism. We have implemented zicIO as both a user library and an OS component, integrating it with four open-source, disk-based databases. Our evaluation demonstrates that all four engines benefit significantly from zicIO's rapid data ingestion capabilities and OS-level page sharing. While we have made progress toward our objectives, there remains room for improvement, particularly with more complex workloads. Nonetheless, we hope that zicIO can benefit DBMSs managing large-scale data processing workloads.

Acknowledgments

We appreciate the reviewers' insightful feedback. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MSIT) (No. 2022R1A2C2008427). Additionally, SK hynix Inc. provided partial support through a research project.

References

[1] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal* 11, 3 (Nov. 2002), 198–215. https://doi.org/10.1007/s00778-002-0074-9

[2] Apache ORC. 2024. Apache ORC File Format. Retrieved October 24, 2024 from https://orc.apache.org/docs https://orc.apache.org/docs.

[3] Apache Parquet. 2024. Apache Parquet File Format. Retrieved July 28, 2024 from https://parquet.apache.org/docs/file-format/ https://parquet.apache.org/docs/file-format/.

[4] Aron, Mohit and Druschel, Peter. 1999. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, New York, NY, USA, 232–246. https://doi.org/10.1145/319151.319167 https://doi.org/10.1145/319151.319167.

[5] Aron, Mohit and Druschel, Peter. 2000. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. *ACM Trans. Comput. Syst.* 18, 3 (aug 2000), 197–228. https://doi.org/10.1145/354871.354872 https://doi.org/10.1145/354871.354872.

[6] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 49–65. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-belay.pdf.

[7] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. 2021. Understanding Host Network Stack Overheads. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/3452296.3472888 https://doi.org/10.1145/3452296.3472888.

[8] 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 *Proceedings of the ACM SIGCOMM 2022 Conference* (Amsterdam, Netherlands) *(SIGCOMM '22)*. Association for Computing Machinery, New York, NY, USA, 767–779. https://doi.org/10.1145/3544216.3544230 https://doi.org/10.1145/3544216.3544230.

[9] George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (Dec. 1960), 655–657. https://doi.org/10.1145/367487.367501

[10] Umur Cubukcu, Ozgun Erdogan, Sumedh Pathak, Sudhakar Sannakkayala, and Marco Slot. 2021. Citus: Distributed PostgreSQL for Data-Intensive Applications. In *Proceedings of the 2021 International Conference on Management of Data* (Xi'an, Shaanxi, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2490–2502. https://doi.org/10.1145/3448016.3457551

[11] DuckDB. 2024. DuckDB Github Repository. Retrieved Oct 20, 2024 from https://github.com/duckdb https://github.com/duckdb.

[12] DuckDB. 2024. DuckDB GitHub repository: Discussion - Can we create an index for external parquet files? #4820. Retrieved Oct 20, 2024 from https://github.com/duckdb/duckdb/discussions/4820 https://github.com/duckdb/duckdb/discussions/4820.

[13] DuckDB. 2024. DuckDB GitHub repository: Discussion - In-memory cache of Parquet data? #6948. Retrieved Oct 20, 2024 from https://github.com/duckdb/duckdb/discussions/6948 https://github.com/duckdb/duckdb/discussions/6948.

[14] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (jul 2023), 2769–2782. https://doi.org/10.14778/3611479.3611486

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076 https://doi.org/10.1145/224056.224076.

[16] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 16, 17 pages. https://www.usenix.org/system/files/osdi20-fried.pdf.

[17] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. *Proc. VLDB Endow.* 16, 9 (may 2023), 2090–2102. https://doi.org/10.14778/3598581.3598584

[18] HammerDB. 2023. HammerDB GitHub project: HammerDB Version 4.6. Retrieved February 28, 2023 from https://github.com/TPC-Council/HammerDB/releases https://github.com/TPC-Council/HammerDB/releases.

[19] HammerDB. 2023. HammerDB GitHub project: HammerDB Version 4.6 - MySQL OLAP Queries. Retrieved February 28, 2023 from https://github.com/TPC-Council/HammerDB/blob/master/src/mysql/mysqlolap.tcl#L957 https://github.com/TPC-Council/HammerDB/blob/master/src/mysql/mysqlolap.tcl#L957.

[20] HammerDB. 2023. HammerDB GitHub project: HammerDB Version 4.6 - PostgreSQL OLAP Queries. Retrieved February 28, 2023 from https://github.com/TPC-Council/HammerDB/blob/master/src/postgresql/pgolap.tcl#L975 https://github.com/TPC-Council/HammerDB/blob/master/src/postgresql/pgolap.tcl#L975.

[21] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. 2020. TiDB: A Raft-Based HTAP Database. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3072–3084. https://doi.org/10.14778/3415478.3415535 https://doi.org/10.14778/3415478.3415535.

[22] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *Proc. VLDB Endow.* 17, 3 (nov 2023), 577–590. https://doi.org/10.14778/3632093.3632117

[23] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 588–604. https://doi.org/10.1145/3477132.3483542 https://doi.org/10.1145/3477132.3483542.

[24] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 113–128. https://www.usenix.org/conference/osdi21/presentation/hwang https://www.usenix.org/conference/osdi21/presentation/hwang.

[25] IBM Corporation. 2022. IBM Db2 version 11.5 documentation: Database Fundamentals; Performance Tuning; Factors Affecting Performance; Query Optimizations; Data Access methods; Scan Sharing. Retrieved February 28, 2023 from https://www.ibm.com/docs/en/db2/11.5?topic=methods-scan-sharing https://www.ibm.com/docs/en/db2/11.5?topic =methods-scan-sharing.

[26] Intel Corporation. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference: RDTSC-Read Time-Stamp Count, pp. 547. Retrieved July 28, 2024 from http://www.intel.com/conten t/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf.

[27] Jens Axboe. 2019. Efficient IO with `io_uring`. Retrieved February 28, 2023 from https://kernel.dk/io_uring.pdf https://kernel.dk/io_uring.pdf.

[28] Jens Axboe. 2022. What's new with `io_uring`. Retrieved February 28, 2023 from https://kernel.dk/axboe-kr2022.pdf https://kernel.dk/axboe-kr2022.pdf.

[29] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conferenc e/nsdi19/presentation/kaffes https://www.usenix.org/conference/nsdi19/presentation/kaffes.

[30] Michael Kerrisk. 2024. Linux manual page - blockdev(8). Retrieved Oct 20, 2024 from https://man7.org/linux/man-pages/man8/blockdev.8.html https://man7.org/linux/man-pages/man8/blockdev.8.html.

[31] Kyoseung Koo, Sohyun Kim, Wonhyeon Kim, Yoojin Choi, Juhee Han, Bogyeong Kim, and Bongki Moon. 2024. PreVision: An Out-of-Core Matrix Computation System with Optimal Buffer Replacement. *Proc. ACM Manag. Data* 2, 1, Article 42 (March 2024), 25 pages. https://doi.org/10.1145/3639297

[32] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518 https://doi.org/10.14778/2367502.2367518.

[33] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-Time Analytical Processing with SQL Server. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1740–1751. https://doi.org/10.1 4778/2824032.2824071 https://doi.org/10.14778/2824032.2824071.

[34] Viktor Leis. 2024. ExMap - Fully explicit virtual memory (exmap) Github Repository. Retrieved Oct 20, 2024 from https://github.com/tuhhosg/exmap https://github.com/tuhhosg/exmap.

[35] Viktor Leis. 2024. Virtual-Memory Assisted Buffer Management (vmcache) Github Repository. Retrieved Oct 20, 2024 from https://github.com/viktorleis/vmcache https://github.com/viktorleis/vmcache.

[36] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *Proc. ACM Manag. Data* 1, 1, Article 7 (may 2023), 25 pages. https://doi.org/10.1145/3588687

[37] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *Proc. VLDB Endow.* 17, 8 (may 2024), 2115–2122. https://doi.org/10.14778/3659437.3659462

[38] 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.*

Association for Computing Machinery, New York, NY, USA, 2530–2542. https://doi.org/10.1145/3448016.3457562 https://doi.org/10.1145/3448016.3457562.

[39] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 399–413. https://doi.org/10.1145/3341301.3359657 https://doi.org/10.1145/3341301.3359657.

[40] MongoDB Inc. 2023. MongoDB's Issue Tracker: Implement asynchronous IO using io_uring API. Retrieved February 28, 2023 from https://jira.mongodb.org/browse/WT-6833 https://jira.mongodb.org/browse/WT-6833.

[41] Oracle. 2021. Heatwave User Guide. https://downloads.mysql.com/docs/heatwave-en.pdf.

[42] Oracle. 2022. Oracle Database 21 Reference: 2.64 Cursor Sharing. Retrieved February 28, 2023 from https://docs.oracle.com/en/database/oracle/oracle-database/21/refrn/CURSOR_SHARING.html#GUID-455358F8-D657-49A2-B32B-13A1DC53E7D2 https://docs.oracle.com/en/database/oracle/oracle-database/21/refrn/CURSOR_SHARING.html#GUID-455358F8-D657-49A2-B32B-13A1DC53E7D2.

[43] Oracle. 2023. MySQL 8.0 Reference Manual: 15.8.6 Using Asynchronous I/O on Linux. Retrieved February 28, 2023 from https://dev.mysql.com/doc/refman/8.0/en/innodb-linux-native-aio.html https://dev.mysql.com/doc/refman/8.0/en/innodb-linux-native-aio.html.

[44] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout https://www.usenix.org/conference/nsdi19/presentation/ousterhout.

[45] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 1–16. https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-peter_simon.pdf.

[46] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780 https://doi.org/10.1145/3132747.3132780.

[47] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[48] Shuveb Hussain. 2024. io_uring: Submission Queue Polling. Retrieved February 28, 2024 from https://unixism.net/loti/tutorial/sq_poll.html https://unixism.net/loti/tutorial/sq_poll.html.

[49] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient Transaction Processing in SAP HANA Database: The End of a Column Store Myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (Scottsdale, Arizona, USA) *(SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 731–742. https://doi.org/10.1145/2213836.2213946 https://doi.org/10.1145/2213836.2213946.

[50] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2021. DBOS: a DBMS-oriented operating system. *Proc. VLDB Endow.* 15, 1 (sep 2021), 21–30. https://doi.org/10.14778/3485450.3485454

[51] The kernel development community. 2015. Reinventing the timer wheel. Retrieved June 3, 2015 from https://lwn.net/Articles/646950/ https://lwn.net/Articles/646950/.

[52] The kernel development community. 2017. Block layer introduction part 2: the request layer. Retrieved November 9, 2017 from https://lwn.net/Articles/738449/ https://lwn.net/Articles/738449/.

[53] The kernel development community. 2023. What is RCU? – "Read, Copy, Update". Retrieved February 28, 2023 from https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html.

[54] The Linux man-pages project. 2023. Linux Programmer's Manual AIO(7). Retrieved February 28, 2023 from https://man7.org/linux/man-pages/man7/aio.7.html https://man7.org/linux/man-pages/man7/aio.7.html.

[55] The PostgreSQL Global Development Group. 2023. Chapter 32. Just-in-Time Compilation (JIT). Retrieved February 28, 2023 from https://www.postgresql.org/docs/current/jit-reason.html https://www.postgresql.org/docs/current/jit-reason.html.

[56] The PostgreSQL Global Development Group. 2023. PostgreSQL 15 github: Heap Access Method Code. Retrieved February 28, 2023 from https://github.com/postgres/postgres/blob/REL_15_STABLE/src/backend/access/heap/heapam.c#L255 https://github.com/postgres/postgres/blob/REL_15_STABLE/src/backend/access/heap/heapam.c#L255.

[57] The PostgreSQL Global Development Group. 2023. PostgreSQL 15.2 Documentation: 20.13.1. Previous PostgreSQL Versions - synchronize sequential scan. Retrieved February 28, 2023 from https://www.postgresql.org/docs/15/runtime-config-compatible.html#GUC-SYNCHRONIZE-SEQSCANS https://www.postgresql.org/docs/15/runtime-config-compatible.html#GUC-SYNCHRONIZE-SEQSCANS.

[58] The PostgreSQL Global Development Group. 2023. PostgreSQL github: Postgres/src/backend/jit/README. Retrieved February 28, 2023 from https://github.com/postgres/postgres/blob/master/src/backend/jit/README https://github.com/postgres/postgres/blob/master/src/backend/jit/README.

[59] Leonard von Merzljak, Philipp Fent, Thomas Neumann, and Jana Giceva. 2022. What Are You Waiting For? Use Coroutines for Asynchronous I/O to Hide I/O Latencies and Maximize the Read Bandwidth!. In *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2022, Sydney, Australia, September 5, 2022*, Rajesh Bordawekar and Tirthankar Lahiri (Eds.). 36–46. http://www.adms-conf.org/2022-camera-ready/ADMS22_merzljak.pdf

[60] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. 2020. F1 Lightning: HTAP as a Service. *Proc. VLDB Endow.* 13, 12 (aug 2020), 3313–3325. https://doi.org/10.14778/3415478.3415553 https://doi.org/10.14778/3415478.3415553.

[61] Irene Zhang, Jing Liu, Amanda Austin, Michael Lowell Roberts, and Anirudh Badam. 2019. I'm Not Dead Yet! The Role of the Operating System in a Kernel-Bypass Era. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 73–80. https://doi.org/10.1145/3317550.3321422 https://doi.org/10.1145/3317550.3321422.

[62] 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 195–211. https://doi.org/10.1145/3477132.3483569 https://doi.org/10.1145/3477132.3483569.

[63] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. 2022. XRP: In-Kernel Storage Functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 375–393. https://www.usenix.org/conference/osdi22/presentation/zhong https://www.usenix.org/conference/osdi22/presentation/zhong.

Received July 2024; revised September 2024; accepted November 2024