AnalyticDB-PG: A Cloud-native High-performance Data Warehouse in Alibaba Cloud

Fangyuan Zhang*
Caihua Yin*
Hua Fan
Alibaba Cloud Computing
Hangzhou, China

Fenghua Fang
Yineng Chen
Xuqi Wang
Alibaba Cloud Computing
Hangzhou, China

Mengqi Wu
Bing Chen
Tianbo Jin
Alibaba Cloud Computing
Hangzhou, China

Sibo Wang
The Chinese University of Hong Kong
Hong Kong, China

Wenchao Zhou
Feifei Li
Alibaba Cloud Computing
Hangzhou, China

{zhangfangyuan.zfy,caihua.ych,guanming.fh,fenghua.ffh,yineng.cyn}
{xuqi.wxq,mengqi.wmq,huanjin.cb,tianbo.jtb,zwc231487,lifeifei}@alibaba-inc.com,swang@se.cuhk.edu.hk

ABSTRACT

In the era of big data, the landscape of data management and analytics has significantly transformed, presenting diverse challenges for cloud platforms. Modern data warehouses face increasing challenges in handling hybrid transactional and analytical processing (HTAP) workloads efficiently in cloud environments. Traditional shared-nothing architectures provide high-performance query execution but suffer from high storage costs and limited elasticity, while shared-storage approaches improve scalability but often struggle with query efficiency due to increased data movement and indexing overhead. Furthermore, existing execution engines lack optimized support for vectorized processing and real-time analytics, limiting their ability to handle large-scale workloads efficiently.

To address these limitations, we introduce AnalyticDB-PG (ADB-PG), a cloud-native, high-performance data warehouse designed for modern analytical workloads. It integrates a unified architecture supporting both Shared-Nothing and Shared-Storage modes, allowing flexible deployment and seamless elasticity. In ADB-PG, we introduce Beam, a hybrid storage engine that efficiently balances row-based and columnar storage for real-time analytics, and Laser, an optimized execution engine leveraging vectorized execution and Just-In-Time compilation to accelerate query processing. The system further incorporates advanced indexing mechanisms, adaptive runtime filtering, and dictionary encoding to enhance performance. Extensive evaluations on TPC-H and TPC-DS benchmarks demonstrate that ADB-PG achieves significant performance improvements while reducing storage and operational costs, making it a compelling solution for modern cloud-based data analytics.

PVLDB Reference Format:
Fangyuan Zhang, Caihua Yin, Hua Fan, Fenghua Fang, Yineng Chen, Xuqi Wang, Mengqi Wu, Bing Chen, Tianbo Jin, Sibo Wang, Wenchao Zhou, and Feifei Li. AnalyticDB-PG: A Cloud-native High-performance Data Warehouse in Alibaba Cloud. PVLDB, 18(12): 5139 - 5152, 2025.
doi:10.14778/3750601.3750633

*Both authors contributed equally to this research.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750633

1 INTRODUCTION

With the proliferation of technologies such as big data analytics [36, 37] and artificial intelligence [38, 58], data volumes are growing exponentially [31], and the demand for complexity of data types processing and real-time data processing is intensifying. However, the limitations of traditional databases in scalability, flexibility, and operational costs make it difficult for them to meet the demands of modern complex business scenarios [1, 15, 35].

In this context, cloud-native databases [13, 18, 24, 26, 30, 56] have emerged as crucial solutions, delivering elasticity, high performance, and cost-effectiveness while handling diverse data workloads [8, 40]. Traditional data warehouse architectures are mainly classified into shared-nothing and shared-storage modes. The shared-nothing mode, used by systems like PolarDB-X [17] and OceanBase [60], offers strong isolation between compute nodes and minimizes inter-node communication overhead, making it ideal for large-scale analytics. However, it suffers from high storage redundancy, limited elasticity, and elevated operational costs due to data replication and rebalancing. In contrast, the shared-storage mode—exemplified by systems that decouple compute and storage using cloud-based object storage—enables on-demand elasticity and cost-efficient data management. Yet, performance bottlenecks remain due to increased I/O overhead, data retrieval latency, and lack of efficient indexing, making it suboptimal for complex analytical queries.

Existing cloud-native data warehouses rely on traditional storage engines and execution frameworks, each with inherent inefficiencies. Many systems use either row-based or column-based storage engines, resulting in suboptimal trade-offs between transactional efficiency and analytical query performance. Column-store engines, while excellent for analytical workloads, struggle with real-time updates due to expensive merge processes and lack of efficient indexing, whereas row-store engines provide fast transactional updates but suffer from slow analytical query performance. Similarly, the execution engines in current systems are often constrained by tuple-at-a-time query processing, which incurs significant overhead due to frequent function calls and lacks support for vectorized execution [29], Just-In-Time (JIT) compilation, and adaptive query optimization. These limitations prevent existing solutions from fully leveraging modern hardware capabilities, resulting in inefficiencies in processing large-scale workloads.

AnalyticDB-PG [3] (abbreviated as ADB-PG), developed by Alibaba Cloud, is designed to address contemporary challenges by delivering a unified, high-performance data warehouse solution supporting both shared-nothing [53] and shared-storage [45] architectures. The journey of ADB-PG began nine years ago. Initially, it was built as a shared-nothing mode using local disk storage to provide efficient analytics services. As public cloud infrastructure [42] matured and new requirements arose, the system evolved to adopt containerization [50] and a compute-storage disaggregated architecture [39], leveraging object storage [43] to enhance elasticity and reduce costs. To serve both on-premise and cloud-based customers, the architecture of ADB-PG was redesigned to abstract the complexities of various storage types, ensuring high data availability and performance in demanding environments.

As data processing demands diversify, providing high performance data processing has become another major challenge for modern data warehouses [25]. First, existing data warehouses lack effective paradigms for handling data updates, resulting in either poor timeliness or low efficiency in real-time data processing. Second, in execution engines different operators have different execution patterns; some are sequential, while others are random. This requires the execution engine to be adaptive, optimizing the execution plan based on the characteristics of operators. To address this, databases employ vectorization techniques and JIT (just-in-time) code generation [9], significantly enhancing execution efficiency by batching data processing operations. For instance, vectorization techniques can process multiple data items in a batch, leveraging modern CPU SIMD (single instruction, multiple data) instructions [29] capabilities to increase data processing throughput. However, despite efficiency improvements offered by current technologies, there are still shortcomings in areas such as storage format management, query optimization, and real-time data processing.

To address these challenges, ADB-PG unifies the benefits of shared-nothing and shared-storage architectures while optimizing execution efficiency. ADB-PG is designed to provide scalability, elasticity, and high-performance query execution by integrating a hybrid storage engine, an optimized execution framework, and advanced indexing techniques. Specifically, ADB-PG introduces Beam, a hybrid storage engine that dynamically balances row and column-oriented storage, ensuring efficient transactional updates while maintaining high-throughput analytical performance. Unlike traditional storage engines, Beam utilizes a combination of log-structured updates, hybrid indexing strategies, and asynchronous data flushing mechanisms to optimize both transactional and analytical query workloads. In parallel, ADB-PG incorporates Laser, an advanced execution engine that leverages SIMD-based vectorized execution, adaptive runtime filtering, and JIT compilation to significantly accelerate query processing. By integrating intelligent storage management, elastic resource scaling, and AI-powered query optimizations, ADB-PG overcomes the fundamental limitations of existing cloud-native data warehouses, delivering high efficiency, cost-effectiveness, and real-time analytical capabilities in cloud environments. Our contributions are as follows:

- We introduce ADB-PG, a cloud-native data warehouse that unifies shared-nothing and shared-storage architectures, optimizing both performance and elasticity.
- We propose Beam, a hybrid storage engine that efficiently balances row-based and column-based storage models, enabling high-throughput analytics with real-time updates.
- We introduce Laser, a high-performance execution engine incorporating vectorized query processing, JIT compilation, and runtime filtering to accelerate analytical workloads.
- Through comprehensive experiments in various aspects, we found that the proposed method has superior performance and cost-effectiveness compared to existing solutions.

2 MOTIVATION

Traditional on-premise databases [59] struggle to meet the evolving demands of cloud computing, which requires flexibility, scalability, and efficient workload management. Cloud infrastructures, spanning both private and public clouds, have driven a shift toward integrated service paradigms to address these challenges. However, earlier versions of ADB-PG faced difficulties in meeting the diverse storage needs of cloud clients, who require scalable, cost-efficient storage with high performance and support for random read/write operations. Conventional storage solutions, while economical, often lack adaptability for broad cloud workloads. For instance, object storage provides scalability and cost benefits but is hindered by its append-only nature, highlighting the misalignment between traditional storage architectures and cloud-native requirements.

Cloud-native data warehouses typically adopt either the Shared-Nothing mode (SNM) or Shared-Storage mode (SSM), each with trade-offs. The SNM partitions data across independent nodes for high-performance queries but suffers from storage redundancy, complex rebalancing, and limited elasticity. The SSM decouples compute from storage for better scalability but incurs high I/O latency and network overhead. A truly unified cloud data warehouse should combine both modes strengths—using local storage for fast access and remote storage for scalability and cost-efficiency—while dynamically optimizing workload distribution. Additionally, cloud-native architectures should fully leverage cloud features like dynamic resource allocation, automatic scaling, and bandwidth expansion, enabling efficient query execution and adaptive performance tuning in modern cloud environments.

Challenges in Storage Engine Design. Early versions of ADB-PG and other cloud-native data warehouses relied on either row-based or column-based storage engines, each with inherent trade-offs. Column-store engines, such as those in Amazon Redshift and Snowflake, enable efficient compression and fast analytical queries but struggle with real-time updates due to high write amplification and complex merge processes. Conversely, row-store engines, as seen in PostgreSQL-based solutions, provide efficient transactional updates but suffer from poor analytical query performance due to inefficient scan operations and higher storage overhead.

Object storage solutions like AWS S3 and Google Cloud Storage, widely used in modern architectures, offer high scalability and cost-effectiveness but have high data access latency, lack native indexing, and handle transactional workloads inefficiently. These challenges require hybrid storage engines that dynamically balance row- and column-based storage, ensuring efficient transactional updates and high-throughput analytical queries.

Challenges in Execution Engine design. Computational frameworks are categorized into compilation execution (e.g., Redshift [30], Spark [61]) and vectorized execution (e.g., Photon [10], Velox [47], ClickHouse [44]). Compilation execution compiles queries into a single optimized kernel, minimizing data materialization overhead but incurring high compilation costs, prolonged startup times, and limited SIMD utilization. Vectorized execution processes data in batches using a columnar memory layout, enhancing SIMD performance and facilitating debugging but introducing additional data materialization overhead. The ideal design should combine these

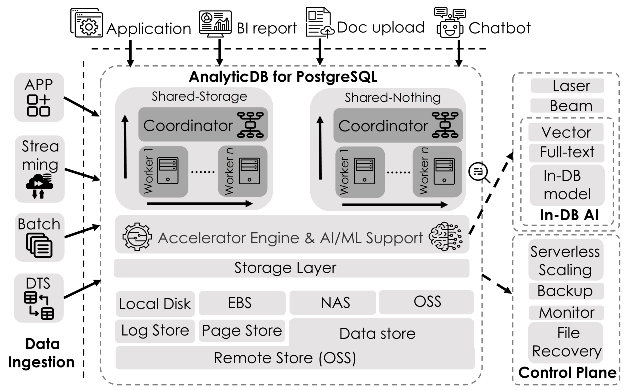


Figure 1: Overall architecture of ADB-PG.

two categories. Reduce function call overhead and optimize the CPU pipeline through batch processing while leveraging JIT compilation to speed up complex expressions. However, in memory-constrained operations such as hash join and sort, JIT compilation does not provide significant advantages over vectorized execution but rather increases development complexity. Balancing performance, flexibility, and development efficiency by selectively applying compilation execution remains a challenge.

3 UNIFIED ARCHITECTURE

3.1 Architecture Overview

ADB-PG Core. Figure 1 illustrates the comprehensive architecture of ADB-PG. On the left, external data is ingested into ADB-PG through various methods, including applications, streaming, batch processing, and DTS [28]. The central framework enables ADB-PG deployment in two distinct configurations: the Shared-Nothing Mode (SNM) and the Shared-Storage Mode (SSM). Similar to other data warehouses with Shared-Nothing architecture, SNM utilizes local disks or cloud disks EBS (Elastic Block Services) on cloud-based virtual machines (ECS). Conversely, SSM employs a storage-separated architecture based on ECS with local caching and remote object storage (e.g., AliCloud OSS, AWS S3, etc.). Regardless of deployment configuration, cluster nodes can scale horizontally (scale-out/in) and vertically (scale-up/down). Unification of both modes is facilitated by a consistent storage layer, which abstracts various storage formats, including local disks, EBS devices, and storage services such as log stores, page stores, and object stores.

From an industry perspective, the evolution of data warehouse architectures has moved from traditional, on-premises deployments based on SNM to cloud-native data warehouses that predominantly adopt SSM architectures. Leading SSM systems (e.g., Snowflake [23], Redshift RA3 [14]) achieve superior elasticity and cost efficiency, but are limited in real-time operations due to object storage. ADB-PG addresses these limitations by enabling both object-storage-based SSM deployments and disk-based SNM deployments through a unified architecture. This design ensures not only the flexibility required by modern cloud scenarios, but also the operational reliability and deployment convenience needed in regulatory or on-premises use cases.

Within the ADB-PG worker nodes, the database kernel houses the Beam storage engine, the Laser execution engine, and the Orca optimizer. These components, along with query optimization techniques, will be examined further in subsequent sections.

In practice, retaining SNM remains essential: (1) Many government and financial customers require on-premises deployment for compliance, where SNM deployment is more cost-effective and avoids the complexity of local object storage; (2) For small to medium data volumes, the extra overhead of SSM components may outweigh storage savings, making SNM preferable; (3) SNM allows data cached locally, providing more stable scan performance; and (4) SNM in ADB-PG, when deployed on ECS/cloud disks, can also leverage cloud-native features like online resizing, offering more flexibility than traditional SN systems.

Furthermore, the right side of the diagram illustrates the interaction with various AI model services. ADB-PG features integrated vector and full-text retrieval capabilities [59], leveraging large language models to enhance intelligent product. This versatile architecture supports the creation of traditional data applications and BI reports, as well as advanced applications like enterprise-specific knowledge-based intelligent Q&A systems and in-database data inference scenarios using vector databases and large models. This part is not the focus of this paper, so we will not go into it in depth.
Control Plane. The control plane ensures security and isolation through the use of Kubernetes (K8S) [4] and RUND [41] secure containers. In enterprise applications, the demands for data security and isolation are critical. ADB-PG employs RUND secure containers to ensure effective isolation among tenants and maintain system security in a multi-tenant environment. For each tenant Pod, ADB-PG provides an additional dedicated network interface card, facilitating secure communication between the control plane and the monitoring plane. This design not only ensures data security but also simplifies the process for administrators to gather operational status information from each Pod. Furthermore, the system implements a strict read-only Volume mounting strategy to confine tenant Pod storage space, thereby mitigating the risk of data breaches between tenants.

Additionally, the control plane can dynamically adjust resource allocations based on workload demands, enabling the serverless functionality of ADB-PG. The elastic design of ADB-PG will be discussed in more detail in Section 3.3.

3.2 Storage Design

HTAP Workload Requirements. We have observed that our clients frequently require both rapid transaction processing and robust analytical operations within their applications. For instance, in logistics and order management scenarios, there is a simultaneous need for real-time order updates and analytical evaluation of metrics such as order completion rates and cancellation ratios. It is difficult to eliminate transactional demands from business processes, necessitating storage designs that support data updates and primary key deduplication. This requirement becomes particularly challenging in scenarios demanding high real-time performance, such as in advertising and marketing. Consequently, developing storage systems capable of supporting Hybrid Transactional/Analytical Processing (HTAP) is particularly demanding.

One of the primary challenges in implementing Storage-Compute Separation in HTAP systems is the complexity of managing indices. The necessity for data updates, along with the demands of full-text and vector searches, means that indices cannot be entirely eliminated from our design. However, the potentially large size of indices presents significant challenges, particularly in a serverless environment where nodes are subject to dynamic changes. In such cases, the cost of migrating state data is considerable.

Unified Storage Layer. We have implemented a unified storage API to abstract the complexities arising from various storage requirements and types, ensuring that ADB-PG can concurrently support both row and columnar storage, as well as local and remote storage solutions. More specifically, the unified storage layer offers the following features:

- It employs a POSIX interface to abstract the intricacies of traditional file systems and object storage, offering a unified API for distributed storage solutions.
- The proposed method can support a hybrid approach to storage types: object storage, which is characterized by low cost, high bandwidth, and high latency with append-only capability, is complemented by a log store for low-latency writes and a page store for low-latency random reads.
- Beyond the existing storage systems, the framework allows for extensibility through plugins to support customized storage solutions on specialized cloud environments, enabling users to tailor their storage strategy without modifying the core codebase of ADB-PG, even in cases where their internal technical infrastructure is not standardized.

By adopting this design approach, we can implement a flexible and maintainable solution using a unified codebase.

To achieve row storage with a compute-storage separation architecture, we employ a combination of a distributed log service (log store), asynchronous log forwarding, and remote page storage (page store). This setup ensures high data availability and consistency. Unlike traditional log services (Log as a Service), our approach supports distributed transaction processing, which provides consistency guarantees for handling multi-partition data updates. Utilizing the aforementioned technologies, our compute-storage separated row storage supports both writing data to object storage (OSS) in an append-only fashion and delivering rapid query performance. ADB-PG's columnar storage takes advantage of object storage, which offers cost efficiency, separation of storage and computation, and scalable capacity.

3.3 Elasticity Design

Elasticity is crucial for the control plane, balancing cost and stability. By leveraging resource pooling and resource isolation, ADB-PG can dynamically acquire storage and computing resources on demand. This subsection details the mechanisms by which ADB-PG achieves elasticity in both storage and computation.

Storage Elasticity. The underlying storage system, EBS, can dynamically adjust I/O operations per second performance, while the OSS can dynamically adjust bandwidth performance. In other words, ADB-PG utilizes the EBS of Alibaba Cloud [20] as the data storage medium, enabling seamless dynamic I/O expansion. During node failure recovery, when an agent in the system detects a node failure, it automatically triggers the expansion of cloud disk I/O to meet the high I/O demands of the recovery process. This ensures that recovery operations do not impact other business operations, maintaining continuous online services and performance stability.

Compute Elasticity. ADB-PG's control plane dynamically allocates resources during peak periods to meet instant computing demands and releases excess resources during troughs, enhancing utilization and reducing costs. For example, under high load or instance failure, the system automatically triggers load balancing and failover to ensure uninterrupted service. When high CPU load is detected, it distributes new tasks to less loaded nodes and redirects business traffic to standby nodes during failures, ensuring an unaffected user experience. In actual deployments, computational elasticity often faces issues: insufficient resources and horizontal scaling can cause brief business interruptions, and new nodes may lack pre-warmed caches, resulting in unstable services during scaling-up. Consequently, users tend to prefer planned elasticity over load-based elasticity.

To address these issues, we optimize our computational elasticity strategy by prioritizing vertical scaling to maximize the stability of business operations during scaling events. We predict business loads before scheduling to ensure that vertical scaling resources are sufficient. In cases where vertical resources are inadequate, we perform cache pre-warming before horizontal scaling. Through these measures, we have reduced the user-perceived downtime during scaling operations by 90%.

4 STORAGE ENGINE BEAM

In modern database and data warehouse systems, data storage models typically include the row-store model, column-store model, and hybrid row-column store model. To enhance data processing efficiency and performance, we propose a hybrid real-time engine called Beam. Unlike traditional hybrid storage systems like SingleStore [48] that maintain durable local state on compute nodes, Beam is designed for completely stateless compute nodes in cloud-native environments, with all durable state externalized to log services and object storage. The proposed storage engine Beam combines the advantages of row storage and column storage models to achieve efficient data read and write performance while enabling instant elasticity and seamless storage-compute separation.

4.1 Beam Overview

The storage engine Beam divides data into two parts: *real-time incremental data (delta)* and *historical full-scale data (primary)*. Delta data adopts the NSM (N-ary storage model) [11] row-store structure, focusing on supporting efficient real-time data insertions, updates, and deletions. Primary data uses the PAX (Partition Attribute Across) [12] hybrid row-column store structure, suitable for high-throughput batch data imports and loading. Moreover, delta data is periodically flushed to primary data based on time and data volume, while primary data is automatically optimized in the background (supporting composite and Z-order clustering [34]) to enhance query and scanning performance. Thus, it avoids duplicating storage space by storing data exclusively in either row or column format. This efficiency is achieved through two distinct data input modes: concurrent streaming and batch import. For each data entry, Beam dynamically selects the appropriate mode based on row count and insertion method (copy or insert). Specifically:

- <u>Concurrent Streaming</u>: Data is initially written into the row storage segment without compression, utilizing shared memory to minimize I/O operations. This approach offers more than a fivefold performance advantage over column storage for typical workload scenarios. This method is preferred for operations involving fewer than 10,000 rows by default. It effectively mitigates the inefficiencies associated with frequent compression and I/O activities observed in pure columnar storage systems like AWS Redshift when handling concurrent real-time writes.

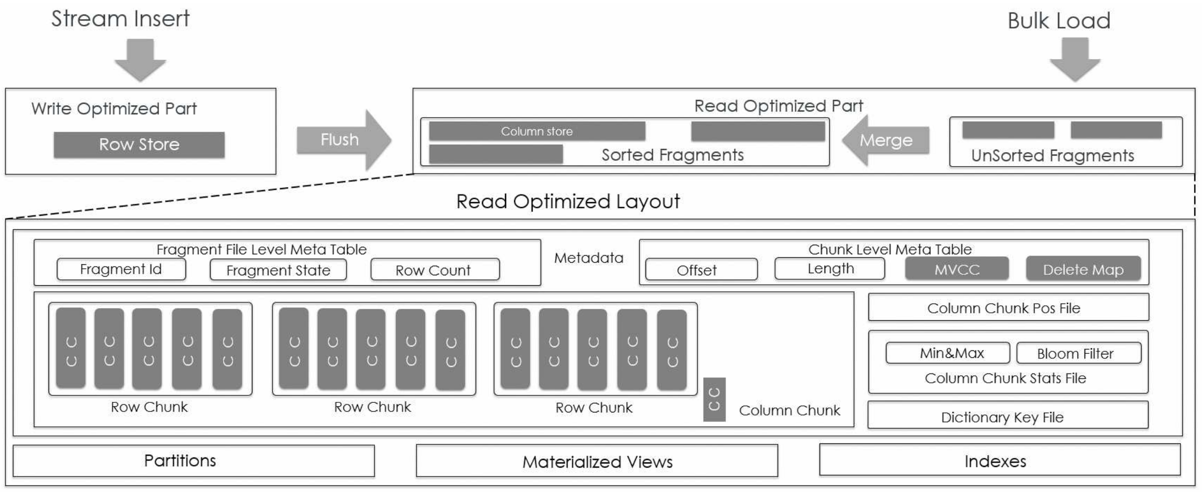
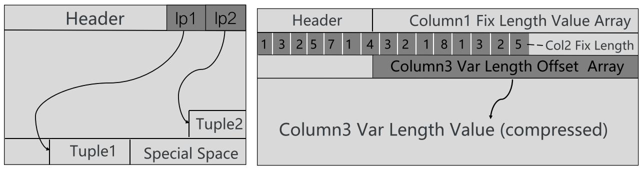


Figure 2: Beam Logical Structure

- *Batch Import*: Data is directed towards the hybrid storage segment, where it accumulates in memory until reaching 64MB, after which it is compressed and stored. Supported compression algorithms include LZ4 and ZSTD; the default LZ4 optimizes for both speed and compression efficiency. Upon completion, an unsorted hybrid storage format file is generated. If the row storage section surpasses a predefined threshold (1/10 of the table size or 64MB), it is automatically offloaded to a new unsorted hybrid form by background processes. When the number of files in the hybrid storage or unsorted files exceeds a preset limit (typically 10), background processes merge, sort, and reclaim storage space occupied by deleted data. The batch import mode is typically selected for volumes exceeding 10,000 rows or during copy imports. This asynchronous, automatic data conversion mechanism prevents redundancy and avoids the storage space inflation seen in systems like Snowflake Unistore, thereby ensuring optimal read performance.

Figure 2 illustrates the logical structure of Beam's storage system. In Beam, a file that uses a hybrid row-column format is referred to as a "Fragment". A Beam table is comprised of fragment families, auxiliary tables, and auxiliary files. Fragments are isolated by write transactions and internally divided into fixed-size segments called Row Chunks. These Row Chunks are logical units where data for each column is stored together as the smallest compression unit, known as a Column Chunk. Unlike data formats such as ORC and Parquet, Beam minimizes interactions with object storage and enhances query throughput by separating auxiliary information into distinct files or tables, rather than embedding it within the data files. The following auxiliary tables/files each serve to implement specific features within Beam.

- *Visibility Auxiliary Table*: This table supports read-committed transaction isolation. Each Beam Row Chunk corresponds to an entry in the visibility auxiliary table, utilizing a BitmapArray to record the visibility of each row within the Row Chunk, known as the delete bitmap. PostgreSQL's native MVCC mechanism ensures transaction isolation. When Beam deletes data, it only marks the corresponding row in the delete bitmap, while updates are handled as delete-plus-insert operations.



Header
lp1
lp2
Tuple2
Tuple1
Special Space
Header
Column1 Fix Length Value Array
1 3 2 5 7 1 4 3 2 1 8 1 3 2 5
Col2 Fix Length
Column3 Var Length Offset Array
Column3 Var Length Value (compressed)

(a) Row Layout of a Page **(b) Column Layout of a Chunk**

Figure 3: Beam Row and Column Layout Examples.

- *Dictionary File*: This file stores dictionary encoding information, which is used for efficient computation (as detailed in Sec. 4.2).
- *Position File*: To avoid inefficient tuple access, each Fragment has an associated Position File. It stores the offsets of data columns within the Fragment using a fixed-length array. When an index retrieves a specific row, it first locates the row's data offset within the Fragment via the Position File.
- *Statistics File*: It facilitates fast scanning by recording metadata such as minimum and maximum values within a Fragment. During scans, the Statistics File helps filter out Column Chunks that do not meet the criteria, reducing the overhead of data retrieval.

Index support. Unlike typical OLAP systems such as RedShift and Snowflake, Beam fully supports secondary indexes available in PostgreSQL, including Hash, Btree, GIN, and Gist. This comprehensive index support is crucial as our clients heavily rely on these indexing mechanisms. Further details on this functionality are provided in subsection 4.2.

4.2 Transaction Processing Support

The support of Beam for transaction processing (TP) derives from its direct row-level data access capabilities, facilitated through its layout design. Leveraging secondary indexes, Beam can achieve rapid point lookups and enforce strong primary key constraints, akin to traditional TP systems. Besides, Beam supports high-concurrency scenarios with row-level locks and upsert semantics.

Direct Row-Level Addressing. Column-based databases commonly utilize row-group storage formats like ORC and Parquet, which complicate direct row access by RowID. Access typically involves consulting metadata to approximate the location, followed by scanning to pinpoint the target row. This challenge is exacerbated in the presence of NULL values, as these formats optimize storage by using null bit arrays to indicate NULLs, necessitating supplementary decoding operations to reconstruct and populate these values during data retrieval.

Beam addresses these limitations through an optimized layout supporting efficient row and column access. In row-based structures (Figure 3a), Beam provides straightforward row-level access comparable to traditional row stores. Even in columnar layouts (Figure 3b), Beam maintains direct row access for both fixed and variable-length columns. The PAX format records NULL positions in bit arrays and fills gaps with zeros, preserving column alignment within row chunks. While introducing minor storage overhead, this is offset by enhanced compression techniques. This design eliminates decoding requirements during reads, significantly improving indexed point query and vectorized execution performance.

The layout enables rapid access to all column values for a row using minimal metadata. Fixed-length columns are accessed via `sizeof<T> * ChunkPosition`, while variable-length types utilize offset arrays. This architecture delivers index point query performance approaching row-based storage levels. Rows are uniquely identified by a 48-bit CTID comprising `<FragmentId, ChunkID, ChunkPosition>`, stored in secondary indexes for efficient data location. Unlike purely columnar models requiring extensive metadata for cross-column row positioning, this approach minimizes overhead. Additionally, physically storing NULL values enables direct computation without decompression in bitpacking scenarios.

Secondary Indexes. It can accelerate query performance by enabling rapid record identification without full scans. While essential for OLTP non-primary key queries and OLAP I/O reduction, traditional columnar storage faces significant overhead during index scans due to cross-column row reconstruction requirements.

Beam leverages its direct row-level addressing to support diverse secondary index types, storing `<Key, RowId>` pairs. Query execution first retrieves the RowId via key lookup, then maps to Beam's `<FragmentId, ChunkID, ChunkPosition>` identifier. The system maintains comprehensive metadata for each `<Fragment, Chunk>`, including row chunk metadata and position files recording column chunk offsets and lengths within fragment files. Row position determination involves accessing `<Fragment, Chunk>` metadata, loading relevant column data, and performing efficient row-level addressing within data blocks to complete index queries.

Strong Primary Key Constraint. Strong primary key constraints ensure uniqueness in OLTP systems but prove costly in OLAP environments due to row-level addressing overhead. Most analytical systems employ weak primary keys, achieving eventual uniqueness through backend compaction rather than real-time enforcement.

Beam implements strong primary key constraints efficiently through its secondary B-tree indexes and row-level locking capabilities. During insertion, Beam verifies primary key existence in the B-tree index and checks deletion status if present. Data insertion proceeds only when the primary key is confirmed absent, ensuring strict uniqueness enforcement.

Row-Level Locking. OLAP systems typically employ chunk-level locking for high-concurrency reads and block-level columnar access. However, this coarse granularity creates substantial overhead for high-frequency single-row updates characteristic of OLTP workloads. Beam achieves superior concurrency for update/delete through row-level locking in columnar storage. The system maintains a dedicated metadata table for rows requiring updates, enabling effective conflict detection and resolution across parallel transactions. During update/delete operations, Beam records deletion-marked row numbers in memory. Concurrent transactions attempting to modify identical rows check for existing row lock markers—if an active transaction is updating the row, subsequent transactions wait; otherwise, they proceed despite sharing the same chunk.

Upsert Support. Beam provides comprehensive Upsert functionality supporting *On Conflict Do Update (OCDU)* and *On Conflict Do Nothing (OCDN)* constructs, enabling user-specified actions when insert operations encounter conflicts.

Upsert differs from standard update/delete operations in that it must handle scenarios where multiple sessions attempt to insert records with the same primary key (pk) that do not yet exist in the table. For example, concurrent sessions inserting `pk=1` into an empty table would both perceive no conflict and proceed, creating duplicate primary keys. Beam addresses this through speculative inserts: data is initially inserted with an index lock. Upon detecting duplicate insertions, later-inserted data is immediately deletion-marked to maintain uniqueness. The system then retries the insertion, entering a wait state if conflicts are detected, until the conflicting transaction commits or rolls back. Unlike alternatives employing *last write wins* strategies that can produce isolation errors in concurrent operations (e.g., `ON CONFLICT(a) DO UPDATE SET b=b+1` potentially yielding incorrect results when both transactions update `b` to 2), Beam prevents such isolation errors, ensuring accuracy in concurrent Upsert transactions.

4.3 Vectorized Scan Optimization

Beam specifically designs and optimizes vectorized scans for the Laser execution engine, offering an advantage that other open-source PAX formats cannot match. Vectorized execution significantly accelerates processing efficiency, particularly when handling multiple columns; it requires row alignment across these columns. As previously discussed, Beam's storage of null values allows the computation engine to perform vectorized calculations directly on decompressed data. This eliminates the need for additional format processing, thereby streamlining execution and enhancing speed.

For workloads involving low-cardinality string scans, filtering, and aggregation, Beam implements global dictionary encoding. Unlike file-level encoding, this approach not only reduces storage requirements for low-cardinality strings but also leverages the capabilities of the execution engine and optimizer to enhance vector and late materialization processes. During queries, data are managed as integers, thereby reducing I/O overhead from extensive data scanning and accelerating operations such as filtering and aggregation. Unlike SingleStore, which supports dictionary encoding solely in the columnar section, Beam optimizes both the delta and base components during queries through dictionary methods. To address inconsistencies in storage models, we introduce a delta dictionary to adapt row storage, where only dictionaries are collected but not encoded. During queries, Beam merges, sorts, and maps the dictionaries from both the base and delta components to form a unified global dictionary.

The performance of Laser is closely linked to the batch processing unit. The storage unit of Beam, the row chunk, serves as the smallest unit for expression filtering and projection calculations in Laser. By unifying the batching units for storage and computation, Beam enhances data computation's memory locality and avoids additional overhead from aligning batch sizes, ultimately boosting batch computation performance.

5 EXECUTION ENGINE LASER

The performance optimization of a data warehouse execution engine significantly impacts overall performance. This subsection discusses operator and expression computation optimizations on the ADB-PG execution engine, Laser. Performance is a critical factor, as it determines whether the response time can meet the requirements of business systems. Additionally, cost is intrinsically linked to performance; a more efficient system can achieve the same business outcomes with fewer hardware resources, thereby reducing costs. Consequently, in data warehouse scenarios involving large volumes of data and complex analytical queries, the performance of the computational layer is of utmost importance. Some computation engine employs a traditional tuple-at-a-time architecture, which suffers from the overhead of frequent interpreted function calls due to the volcano model. This approach cannot also leverage advanced CPU features such as SIMD instructions, leading to suboptimal performance. Therefore, we developed the high-performance computing engine Laser to deliver ultra-fast analytical server. Unlike existing systems that statically choose execution strategies, Laser introduces an adaptive hybrid approach that analyzes each query fragment at runtime to select between vectorized batch execution and JIT-compiled kernels. Laser addresses these challenges through various optimizations focusing on the computational framework, memory layout, and expression evaluation. These optimizations are co-designed with Beam chunked storage format, ensuring aligned batch units and memory layouts for maximum efficiency. Moreover, it incorporates two key performance-enhancing features: runtime filter and dictionary encoding with cost-based plan adaptation.

5.1 Computational Framework

The computational framework in the industry predominantly fall into two categories: Compilation Execution, exemplified by systems like Redshift [30] and Spark [61], and Vectorized Execution, represented by Databricks Photon [10], Velox [47], and ClickHouse [44]. Comparing with these, Laser dynamically combines both strategies based on workload-aware heuristics, enabling superior performance for diverse query patterns in cloud-native workloads. The advantages and disadvantages of these approaches are:

- *Compilation Execution:* This runtime compilation process can transform the entire computation into a single kernel, executed within CPU registers, yielding significant performance benefits without additional data materialization overhead. However, the compilation process incurs unavoidable costs, potentially prolonging compile times in scenarios with complex computations but small data volumes. Additionally, this approach poses challenges in development and debugging, and fails to fully exploit CPU SIMD capabilities.
- *Vectorized Execution:* This method processes data in batches, storing data within each batch in a columnar memory layout, thus maximizing SIMD performance enhancements. Code written in C++ facilitates easier debugging and development and incorporates partial adaptive execution capabilities. However, this approach introduces additional data materialization overhead, requiring intermediate computation results to be repeatedly read and written in memory.

Laser integrates the strengths of both frameworks, primarily adopting Vectorized Execution, while combining Compilation Execution with Vectorized Execution for expression computation. The corresponding framework is shown in Figure 4. The rationale behind this choice is that the core of Vectorized Execution—batch-at-a-time computation—allows the CPU to focus on continuous data processing, significantly reducing interpreted function call overhead. The column-oriented memory layout for batch computations ensures that continuous storage and processing of similar data optimally leverage CPU pipelining and SIMD instructions. Meanwhile, Compilation Execution offers significant advantages in complex computation scenarios by optimizing instruction sets and reducing function call overhead. This is particularly beneficial in intricate expression computations. In other scenarios, such as hash joins and sorting, which are typical memory-stall scenarios, JIT approaches offer no distinct advantages over Vectorized Execution and present considerable development challenges. Section 5.3 will details the experssion evalution using this framwork.

5.2 Memory Layout

Firstly, we provide a concise overview of two memory layouts:

- **DSM (Decomposition Storage Model)**, which aggregates and stores data from the same field within records. This memory layout is well-suited for scenarios involving continuous data access and computation.
- **NSM (N-ary Storage Model)** aggregates and stores data from all fields of each record. This model is appropriate for scenarios where data is accessed randomly in memory yet requires computations across all fields.

Database computation scenarios can be categorized into two main types based on memory access patterns:

Sequential Memory Access: This category includes operations like scans and shuffles, characterized by computations running on contiguous memory. For such scenarios, a column-oriented Data Storage Model (DSM) is the optimal choice, as it naturally facilitates high-performance computing through SIMD instruction sets directly on the same data memory.

Random Memory Access: This category encompasses operations such as aggregation, join, and sort, where the primary bottleneck is memory latency due to cache misses. If a DSM memory layout is used in such situations, operations like join key comparisons and projection outputs will experience frequent cache misses due to the separate storage of each column. Therefore, for these scenarios, a Row-oriented Storage Model (NSM) is ideal. However, not all data is stored using the NSM structure. For instance, in hash join, join keys and columns involved in filtering and projection calculations are organized individually in NSM format. This approach ensures that the overall algorithm adheres to a batch-vectorized paradigm, while simultaneously minimizing cache misses at each computational stage by leveraging the NSM memory layout.

5.3 Expression Evaluation

In Sec. 5.1 we mentioned the use of an adaptive hybrid approach combining compilation and vectorized execution for expression computation. This subsection delves into the optimal scenarios for each technique and explains how adaptive selection is achieved.

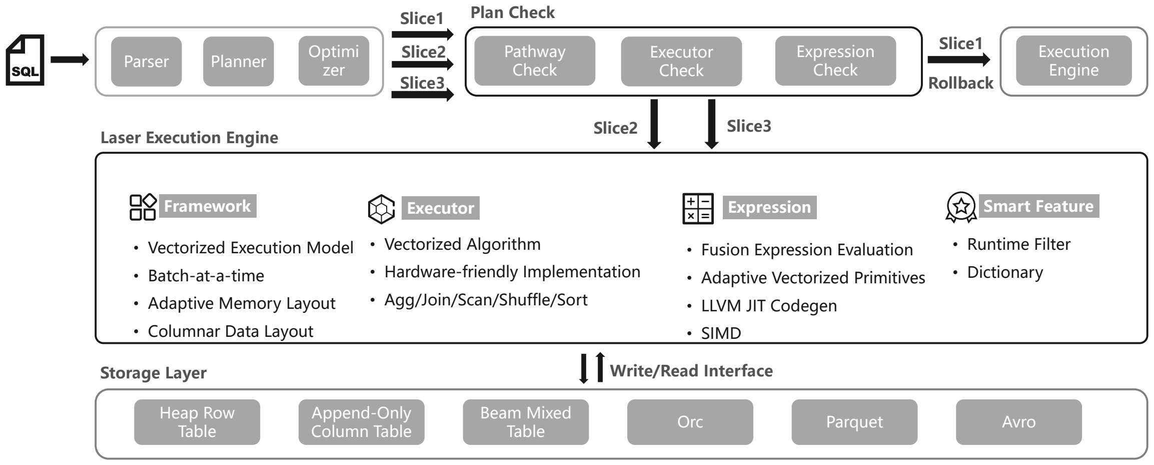


Figure 4: Laser Logical Structure

- Expression Evaluation with Vectorized Execution: SIMD Utilization: For arithmetic or comparison operations in a column-oriented memory layout, Vectorized Execution fully exploits modern CPU SIMD features (e.g., AVX2, AVX512), achieving 2–8× higher computational performance. Adaptive Execution: Batch processing enables dynamic simplification of computational logic based on null value presence during expression evaluation. Short-Circuit Scenarios: Logical operators like AND/OR or CASE WHEN may short-circuit; frequent short-circuit checks can introduce extra overhead.
- Expression Evaluation with Compilation Execution: (1) Integrated Kernel Functions: By compiling all computations into a single kernel at query runtime, calculations are efficiently executed in registers. In short-circuit scenarios, results are returned immediately when conditions are met, unlike Vectorized Execution, which requires repeated evaluations. (2) Compilation Overhead: Compiling an expression typically takes about 10ms. For complex cases with small data volumes, compilation time can exceed execution time, degrading performance. Additionally, row-by-row computation does not utilize SIMD instructions, limiting performance for continuous data processing scenarios.

Generally, Vectorized Execution excels in scenarios involving continuous computations, while Compilation Execution is more efficient in short-circuit scenarios with logical operations. The adaptive approach we employ in Laser integrates these two techniques, automatically routing to the optimal performance method based on context. Specifically, in column-oriented memory layout scenarios, we determine at runtime whether expression evaluation involves short-circuit computations. If so, we dynamically decide whether to compile a kernel based on the data volume, thereby accelerating expression evaluations in large-data scenarios while avoiding unnecessary compilation costs in smaller datasets.

Moreover, as introduced in the memory layout section, hash joins or aggregations that use a row-oriented method require row-based computation for projection outputs. Under such circumstances, Vectorized Execution would incur function call overhead due to processing one row at a time within loops. Therefore, using Compilation Execution provides a performance advantage in these cases.

JIT (Just-In-Time) compilation and vectorization (vectorized primitives) are two distinct optimization techniques, each with its own advantages and suitable contexts.

JIT compilation performs runtime compilation, allowing for dynamic optimization based on actual data and execution environments, potentially offering better performance than static compilation. However, since compilation occurs during runtime, there may be initial execution delays. JIT compilation is particularly well-suited for scenarios where runtime information is critical for optimization, such as large-scale data processing.

Modern processors often include SIMD instruction sets, enabling vectorization to fully exploit these hardware features by processing multiple data elements in parallel. For large arrays or matrix computations, vectorization can significantly enhance performance. By utilizing a loop function approach to reduce the overhead of function calls, SIMD instruction sets can further enhance performance when applied to columnar memory layouts. However, vectorized primitives require structured data processing and lack the flexibility of loops for complex conditional logic. Understanding and applying vectorization necessitates a deep comprehension of array operations and underlying hardware. In summary, we employ vectorized primitives for expression computations on DSM data structures, while JIT compilation is preferred for NSM structures.

5.4 Optimization

Next, we introduce two high-performance features designed for specific scenarios: runtime filter and dictionary encoding. Additionally, it explains how the computational engine synergizes with storage and the optimizer to deliver exceptional analytical performance.

Runtime filter. HashJoin is a widely used and important database operator. When join selectivity is low—meaning the probe side output is much smaller than its input—a runtime filter can greatly reduce data volume, I/O and network overhead. This is achieved by building a Bloom Filter from the HashJoin building and pushing it down to the scan, shuffle, and computation nodes on the probe side. Most analytical engines support runtime filters, typically adding them after execution plan generation based on estimated join selectivity from statistics. However, late introduction can change output data volumes and potentially lead to suboptimal join orders.

To solve this, we implement a runtime filter enforcer in our cascade optimizer. This allows the optimizer to update cardinality and cost estimates immediately after introducing runtime filters, ensuring better join order decisions. In our system (ADB-PG), Runtime Filter requirements are generated at the Enforcer stage, propagated down the operator tree, and materialized as physical runtime filter operators. Along this path, we dynamically adjust cardinality and cost, letting the optimizer select plans that account for the impact of runtime filters. For example, in a TPC-H query involving lineitem, orders, and supplier tables, two join plans for lineitem and orders are possible. By updating cardinality using runtime filter requirements, the optimizer identifies that lineitem filtered data will be smaller than orders, and correctly places lineitem as the hash build (inner) side, achieving optimal join reordering.

```
select *
from (select l_suppkey, sum(l_quantity) as sum_qty
      from lineitem,
           orders
      where l_orderkey = o_orderkey
      group by l_suppkey) t,
     supplier
where l_suppkey = s_suppkey
  and s_comment like '%Customer%Complaints%';
```

Besides, the effectiveness of the Bloom Filter is determined by its size, which directly influences its filtering capability. Relying purely on estimated cardinality could lead to suboptimal Bloom Filter sizes: too small, leading to poor filtering, or too large, increasing space usage and network transmission, thereby degrading performance. In Laser, the algorithm finalizes the data materialization on the build side before constructing the hash table, allowing for the precise calculation of the Bloom Filter's optimal size. This ensures an ideal balance between filtering effectiveness and resource utilization.

Dictionary Encoding Dictionary encoding is another widely used performance-enhancing feature, particularly effective in scenarios involving computations with low-cardinality string data, such as string filtering, sorting, and grouping. In these computations, converting string operations into integer operations via dictionary encoding can significantly boost performance.

The principle behind dictionary encoding involves encoding string data as integer values within each fragment file at the storage layer. A corresponding dictionary file stores the mapping information. During the plan generation phase, the optimizer collects dictionary encoding requirements in a top-down manner through the Plan Tree and matches these requirements with the dictionary capabilities of the underlying scan operation. Plans that satisfy these requirements undergo node rewriting, with a Decode operator inserted at the end, adhering to the Late Decode principle.

Within the computation phase, the Laser engine consolidates dictionary information from multiple fragments during scanning to create a unified dictionary. It maps fragment-specific dictionary data to the global dictionary, producing uniformly encoded integer values. This facilitates high-performance computations using these encoded integer values. The final step involves decoding the integer values back into strings at the decode node.

6 EVALUATION

In this section, we evaluate the performance of ADB-PG across various aspects, including the following problems:

- What is the overall performance and cost of ADB-PG?
- How does Beam balance and enhance the efficiencies of both row-oriented and column-oriented storage?
- How does Laser improve analytic performance of ADB-PG?
- How effective is resilience in the presence of failures?

Case	vCPU	Configuration TW=Transaction Worker AW=Analytic Worker	Transactional Throughput (TpmC)	Analytical Throughput (QPS)
1	16	0 TWs, 2 AWs	-	0.14
2	16	16 TWs, 0 AWs	7932	-
3	16	16 TWs, 2 AWs sharing one workspace	3260	0.061

Table 1: Summary of ADB-PG CH-BenCHmark results (1000 warehouses, 20 minute test executions)

6.1 Overall Performance

We test the proposed system scalability using different data sizes. A high-performance server is utilized to ensure the accuracy and validity of our test results. The hardware configuration consists of eight compute nodes, each with 12 cores, leading to a total of 96 cores, leveraging Alibaba Cloud virtualization technology. Specifically, the servers are powered by Intel Xeon Platinum 8369B (Ice Lake) processors, which have a base clock speed of 2.9 GHz and a maximum turbo frequency of 3.5 GHz across all cores. The system is equipped with 768 GB of RAM and utilizes PL1 level cloud storage with Elastic Block Storage. For the software environment, the experiments were conducted on AliOS 8.

Query performance. For an analytical data warehouse, achieving superior query performance is a core capability. We compare the TPC-H query execution times of ADB-PG using shared-storage mode (SSM) and shared-nothing mode (SNM). Figure 5 shows the performance of these two architectures across various queries. The geometric mean of all queries is also reported, as recommended in the official documentation [5]. We observe that ADB-PG with SSM delivers query performance comparable to SNM overall. For some queries (e.g., Q1, Q18), SSM demonstrates significant performance advantages. This is because, even with a unified storage API, ADBPG-SSM enables multi-shard optimization, making it feasible to deploy twice as many nodes under the same scenario. Therefore, for scan-intensive queries, SSM is faster than SNM when the cache is hit. However, for certain queries (such as Q4, Q5), SSM performs slightly worse than SNM due to the overhead of deploying more nodes. For queries requiring network shuffling, this leads to increased network overhead, degrading performance for SQL operations involving substantial data transfer. In summary, SSM delivers superior performance for most queries and, given its significant cost savings, ADB-PG with SSM demonstrates clear advantages.

Monetary cost. We compare the monetary costs associated with different data volumes under the SSM and the SNM, as shown in Figure 6. The results indicate that the SSM achieves a reduction in costs by an order of magnitude compared to the SNM. This is primarily due to the innovative design of the SSM, which reduces redundant components and improves resource utilization. By employing a unified storage layer, the SSM allows for the deployment of OSS at a lower cost, thereby reducing the overall system expenses. Furthermore, as the volume of data changes, the SSM demonstrates a significant advantage in terms of cost. This provides practitioners with strong guidance in selecting a data warehouse.

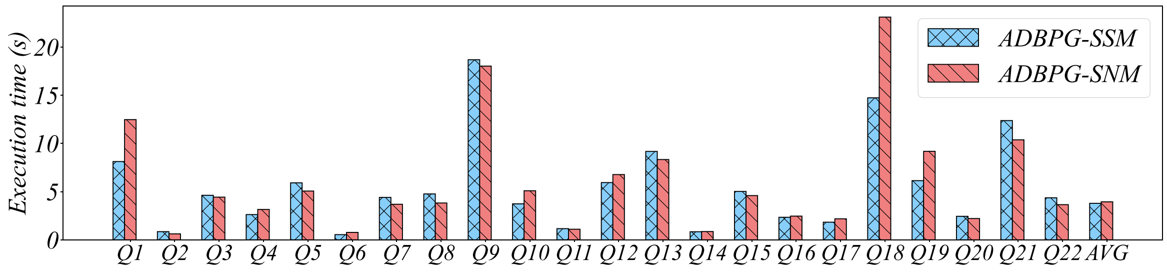


Figure 5: The time cost of different queries on 1TB TPC-H.

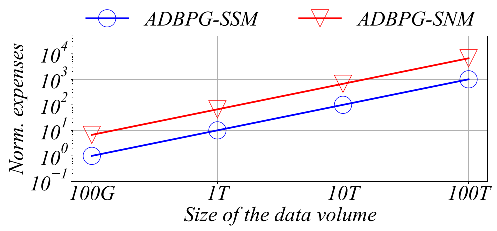


Figure 6: The monthly cost of different data sizes.

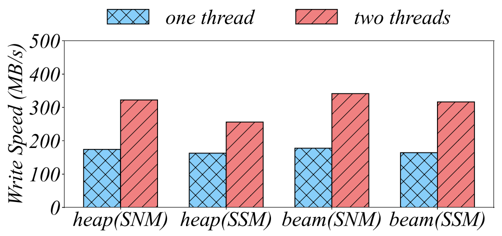


Figure 7: Comparison of write speeds of different strategies.

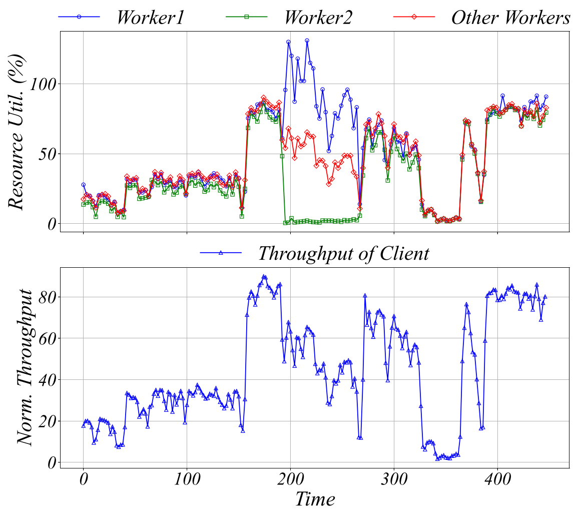


Figure 8: CPU utilization and throughput of the client over time when a failure occurs.

Write performance. Figure 7 displays the consumed time for the Heap and Beam tables during the copy import of a 100GB TPC-H dataset under both the SSM and the SNM. The results indicate that with one concurrent import, both the Heap and Beam tables show a performance difference of approximately 10% between the elastic storage and the decoupled storage-compute modes during bulk writes. This difference can be attributed to the architectural changes, which lead to additional resource consumption during the write operations, thereby affecting the write performance by around 10%. In the case of two concurrent imports, the Beam table maintains consistent bulk write performance across both modes, indicating that the impact of decoupled storage-compute on write efficiency is stable. However, for the Heap table, the performance gap increases under 2 concurrent imports due to IO bottlenecks generated by the page store.

Mixed workloads. To evaluate ADB-PG on mixed workloads, we run CH-BenCHmark [21] with various configurations by adjusting the number of transactional workers (TWs) and analytic workers (AWs) on shared tables, as summarized in Table 1. As defined by the benchmark and related work [48], we test and report the transaction throughput (TpmC) and analytical throughput (QPS). In test case 1, running 16 TWs alone produced the highest transactional throughput (7932 TpmC). In test case 2, with only 2 AWs, yielded the highest analytical throughput (0.14 QPS). For test case 3, when 16 TWs and 2 AWs run concurrently within a single workspace, both experience approximately half throughput degradation compared to running in isolation, indicating balanced resource sharing without severe interference. If each workload were executed in a workspace with dedicated CPUs, the aggregate throughput would approach the sum of scenarios 1 and 2 due to the elimination of resource contention. These results demonstrate that ADB-PG achieves both high throughput for isolated workloads and fair performance in mixed HTAP scenarios through effective resource management.

6.2 Beam

In this section, we evaluate the batch write and range query performance of Beam, benchmarking it against Heap, the original row-oriented storage of PostgreSQL [7], and the columnar storage from Amazon Redshift [30]. All experiments were conducted on database clusters of equivalent sizes. The ADB-PG cluster is configured with eight compute nodes, each having 2 vCPUs and 16 GB of memory. In comparison, we used an ra3.large Amazon Redshift cluster as the counterpart for these experiments.

Batch write performance. We demonstrate the advantage of using Beam for batch writes on the TPC-H `lineitem` table with varying batch sizes. Figure 9 shows the rows per second (RPS) for Heap (blue bars), Beam (red bars), and Amazon Redshift's columnar storage (green bars) across batch sizes from 1 to 50,000. The results highlight Beam's advantage in dynamically choosing between the write-optimized row store and bulk loading into the columnar store. With small batches, both Beam and Heap outperform Redshift due to the row store's efficient stream insert. However, Beam incurs overhead from writing column chunk pos and dictionary key files,

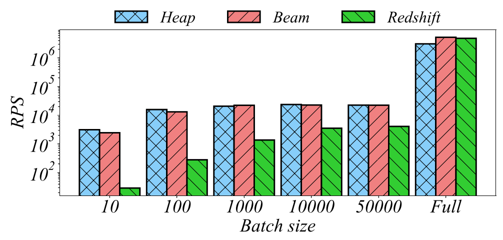


Figure 9: Comparison of RPS of different batch size of write.

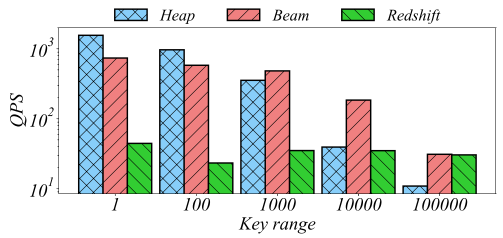


Figure 10: Comparison of QPS of different key ranges of range queries.

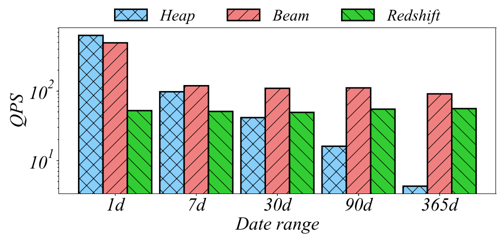


Figure 11: Comparision of QPS of different date ranges of range queries with zone map.

so its RPS is slightly lower than Heap's at batch sizes 10 or 100. As batch size approaches table scale (i.e., using `INSERT INTO SELECT * FROM`), labeled "Full" in Figure 9, both Beam and Redshift outperform Heap by leveraging columnar storage efficiency.

Range query performance. We demonstrate the advantages of using Beam for range queries by executing SQL statements on the TPC-H table `lineitem`, which comprises 60 million rows. The SQL statements are structured as `SELECT COUNT(1) FROM lineitem WHERE l_orderkey IN RANGE [l,u]`. The variables `l` and `u` represent randomly generated lower and upper limits of the key ranges, with varying gaps between them. Figure 10 presents the queries per second (QPS) performance for Heap, Beam, and columnar storage from Amazon Redshift, evaluated across different range gaps ranging from 1 to 100,000. The results highlight Beam's advantage in dynamically choosing between index searches and sequential scans based on the range gaps. When the key range is small, both Beam and Heap outperform Amazon Redshift's columnar storage due to their effective utilization of index searches. However, compared to Heap, Beam incurs additional overhead from data decompression, leading to a noticeable decline in QPS. As the key range expands, Beam demonstrates superior performance compared to Heap because it offers sequential scan capabilities.

We further assess range query performance using SQL statements structured as `SELECT COUNT(1) FROM lineitem WHERE l_shipdate IN RANGE [l,u]`, where `l` and `u` represent date ranges. Both Beam and Amazon Redshift use the zone map to enhance the sequential scan performance. Figure 11 presents the queries per second (QPS) performance for Heap, Beam, and columnar storage from Amazon Redshift across different date ranges. The results indicate that both Beam and Redshift outperform Heap when the date range extends beyond seven days, as a sequential scan with zonemap execution plan is employed. Although the sequential scan performance does not further improve as the date range grows, Beam continues to outperform Redshift due to system optimizations achieved through Just-In-Time (JIT) compilation.

6.3 Laser

6.3.1 Dictionary Encoding. In this section, we present experimental results for laser's performance feature, including runtime filter and dictionary encoding. The test platform is a cluster composed of 48 computation nodes, each node use ecs.r7.large with 2 core vcpus, 16GB RAM and 200GB PL1 ESSD devices.

6.3.2 runtime filter. Table 3 shows the results from performance tests conducted on TPC-DS queries Q24 and Q64 using a 1TB TPC-DS dataset in an ADB-PG environment with 48 computing nodes. The scenarios tested include disabling runtime filter, enabling runtime filter without the join reorder capability, and enabling both runtime filter and join reorder. The results show that applying a runtime filter to pre-select data significantly reduces execution time. Furthermore, by incorporating the runtime filter at the Enforcer stage and utilizing a cost model to select more optimal plans considering the runtime filter, the query execution structure can be further optimized, leading to additional time reductions.

To evaluate the effectiveness of the Adaptive Runtime Filter Size, we conducted tests using the 1TB TPC-H Benchmark. The test environment consisted of 48 compute nodes, and we focused on analyzing the performance of SQL queries Q9 and Q12 with and without the adaptive capability in the Runtime Filter. The results are presented in Table 4.

In the case of Q9, two runtime filters were applied. With adaptive sizing enabled, the filtering ratios improved from 56% and 71.6% to 77.16% and 94.56%, respectively, reducing the runtime from 21.28 seconds to 17.65 seconds. Q12 presented a different scenario where the estimated size significantly exceeded the actual size. With adaptive sizing, the filtering ratio changed from 97.82% to 94.57%. Although the filtering ratio decreased, the actual execution time also lowered because the HashTable actual row count was 650,000, compared to an estimated 33.55 million rows. An excessive size leads the Bloom Filter to require more time during probe operations and memory allocation. Experimental results demonstrate that adaptive runtime filter can determine optimal bloom filter bit array size based on precise runtime information, thereby preventing performance degradation due to substantial estimation errors.

Table 2 presents experimental data on the effectiveness of dictionary encoding using the standard SSB performance benchmark dataset at a scale of 1TB. The tests were conducted on an ADB-PG cluster with 48 computing nodes, focusing on various queries in conjunction with dictionary encoding features. The column p_brand represents a low-cardinality string column with an actual cardinality of 1,000. The comparison highlights the performance impacts of using dictionary encoding versus non-dictionary encoding under different query conditions.

Table 2: Performance Impact of Dictionary Encoding (DE) in SSB Queries

Description	SQL	DE	No DE
Q1 Equal Filter	SELECT count(*) FROM lineorder_flat WHERE p_brand = 'MFGR#12';	0.21s	1.23s
Q2 Range Filter	SELECT count(*) FROM lineorder_flat WHERE p_brand >= 'MFGR#200' AND p_brand <= 'MFGR#400';	0.95s	38.87s
Q3 Group Key	SELECT s_brand, count(*) FROM lineorder_flat GROUP BY s_brand;	1.56s	6.97s
Q4 Agg Expression	SELECT s_region, SUM(CASE WHEN p_brand >= 'MFGR#2221' AND p_brand <= 'MFGR#2228' THEN lo_extendedprice ELSE lo_ordtotalprice END) FROM lineorder_flat GROUP BY s_region;	3.36s	52.36s

Table 3: Performance Impact of Runtime Filters(RF) on TPC-DS Queries

Query	No RF	RF	RF + Enforcer
TPC-DS Q24	6414 ms	3224 ms	3138 ms
TPC-DS Q64	10275 ms	7687 ms	6047 ms

Table 4: Performance Comparison of Adaptive and Normal Runtime Filters(RF) on TPC-H Queries

Query	Normal RF	Adaptive RF
Q9	21.28s	17.65s
Q12	3.01s	2.81s

6.4 Recovery

To emulate failover behavior, we artificially induce a crash failure on one worker node during live workload execution. Figure 8 shows the changes in CPU utilization across different worker nodes and client throughput over time in ADB-PG under a real workload case. In this scenario, Worker 1 serves as the backup node for Worker 2. Upon the unexpected failure of Worker 2, we recorded the automatic failover to Worker 1, accompanied by an *automatic* scale-up. The upper part of the figure displays the CPU utilization for Worker 1, Worker 2, and Other Workers, where Other Workers indicate the average resource utilization of other nodes. Before the failure, the resource utilization across all nodes remained consistent. After the failure occurred, the utilization of Worker 2 dropped to zero, causing the resource utilization of Worker 1 to immediately increase to ensure that the average resource utilization between Worker 1 and Worker 2 remained consistent with that of the other nodes. This guarantees the resource utilization of the overall system remains stable. The lower part of the figure shows the normalized client throughput. We observe that throughput changes along with resource utilization across all working nodes. After the failure of Worker 2, the throughput continues to vary in line with the resource utilization of other workers, indicating that the system can still operate stably amid load fluctuations and sudden failures.

7 RELATED WORK

Cloud-Native databases. Cloud-native databases leverage cloud scalability and resilience, supporting microservices, dynamic scaling, and high availability. Examples include Amazon Aurora [57], Google Cloud Spanner [22], and MongoDB Atlas [2]. These databases utilize containerization and orchestration, enhancing deployment and management, while incorporating automated backup, monitoring, and security. Modern systems like Azure Cosmos DB [52] and CockroachDB [55] support diverse workloads with features such as multi-region replication, real-time analytics, and serverless operation, evolving with cloud adoption.

Shared-nothing and shared-storage architectures. Shared-nothing databases, used in systems like PolarDB-X [17], OceanBase [60], and TiDB [33], partition data across independent nodes for horizontal scalability but face challenges in coordination and fault tolerance. In contrast, shared-storage architectures, exemplified by Oracle RAC [19], enable easier coordination but suffer from high network overhead and limited cloud flexibility. Modern cloud-native databases [27] explore multi-primary configurations, improving scalability yet introducing performance trade-offs due to log synchronization inefficiencies and concurrency control complexities.

Mixed row/column formats. Mixed row/column formats support HTAP workloads by combining row-based write speed with columnar analytics. There are actually many differences among different products. Systems like SingleStore [48] and CockroachDB [54] tightly couple compute and storage with durable local state. Others, like TiDB [32] and AlloyDB [6] separate compute from a stateful, replicated storage cluster. Snowflake [23] further disaggregates by using immutable cloud storage. In sharp contrast, the cloud-native design of Beam in ADB-PG uses completely stateless compute nodes by externalizing state to an external log service for fast commits and to object storage as the ultimate source of truth. This fully disaggregated model provides superior elasticity and fault tolerance while preserving low-latency transactional performance.

Execution Engine. Vectorization Execution Engine, used in system as DuckDB [49], ClickHouse [51], and Databricks Photon [16], achieve acceleration by combining batch computing and columnar computing, making full use of the characteristics of CPU SIMD. The compilation execution, as Redshift [30], UmbraDB [46], through the Just-in-time Compilation technology, dynamically generates a kernel based on actual running SQL, and achieves high-performance computing by eliminating the overhead of function calls and low-latency access to data in registers. In contrast, Laser uniquely employs adaptive mode switching, dynamically choosing between vectorized and compiled execution at runtime for each fragment. Furthermore, Laser tightly couples its execution strategies with storage layout and runtime optimization, enabling deeper integration of runtime filters and dictionary encoding for adaptive plan optimization, which is beyond the capabilities of traditional engines.

8 CONCLUSION

In this work, we presented ADB-PG, a cloud-native, high-performance data warehouse that integrates shared-nothing and shared-storage architectures to optimize scalability and elasticity. Our system introduces Beam, a hybrid storage engine, and Laser, a high-performance execution engine, enabling efficient analytical query processing. Extensive evaluations on industry-standard benchmarks demonstrate that ADB-PG achieves superior performance and cost-efficiency compared to existing solutions.

REFERENCES

[1] 2012. Data, data everywhere. *The Economist* (2012).

[2] 2022. MongoDB Atlas. https://www.mongodb.com/cloud/atlas/.

[3] 2024. AnalyticDB for PostgreSQL. https://www.alibabacloud.com/en/product/hybriddb-postgresql.

[4] 2024. Kubernetes. https://kubernetes.io/.

[5] 2024. THE TRANSACTION PROCESSING COUNCIL. http://www.tpc.org/tpch/.

[6] 2025. AlloyDB for PostgreSQL. https://cloud.google.com/alloydb.

[7] 2025. PostgreSQL. https://www.postgresql.org/.

[8] Daniel Abadi, Anastasia Ailamaki, David G. Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter A. Boncz, Surajit Chaudhuri, Alvin Cheung, AnHai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Y. Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh M. Patel, Andrew Pavlo, Raluca A. Popa, Raghu Ramakrishnan, Christopher Ré, Michael Stonebraker, and Dan Suciu. 2022. The Seattle report on database research. *Commun. ACM* 65, 8 (2022), 72–79.

[9] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. 1998. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *PLDI*. 280–290.

[10] Sameer Agarwal, Paul Barham, and Ion Stoica. 2022. Photon: A Fast Query Engine for Lakehouse Systems. *PVLDB* 15, 12 (2022), 3665–3678.

[11] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. 2002. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.* 11, 3 (2002), 198–215.

[12] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *VLDB*. 169–180.

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

[14] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J Green, Monish Gupta, Sebastian Hillig, et al. 2022. Amazon Redshift re-invented. In *SIGMOD*. 2205–2217.

[15] Alejandro Vera Baquero, Ricardo Colomo Palacios, and Owen Molloy. 2016. Real-time business activity monitoring and analysis of process performance on big-data domains. *Telematics Informatics* 33, 3 (2016), 793–807.

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

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

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

[19] Sashikanth Chandrasekaran and Roger Bamford. 2003. Shared Cache - The Future of Parallel Databases. In *ICDE*. 840–850.

[20] Alibaba Cloud. 2022. Elastic Compute Service Block Storage. https://www.alibabacloud.com/blog/what-is-elastic-block-storage_597401.

[21] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems*. 1–6.

[22] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8.

[23] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *SIGMOD*. 215–226.

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

[25] Nedim Dedic and Clare Stanier. 2016. An Evaluation of the Challenges of Multilingualism in Data Warehouse Development. In *ICEIS*. 196–206.

[26] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, and Yongjun He. 2020. Taurus Database: How to be Fast, Available, and Frugal in the Cloud. In *SIGMOD*. 1463–1478.

[27] Alex Depoutovitch, Chong Chen, Per-Åke Larson, Jack Ng, Shu Lin, Guanzhu Xiong, Paul Lee, Emad Boctor, Samiao Ren, Lengdong Wu, Yuchen Zhang, and Calvin Sun. 2023. Taurus MM: bringing multi-master to the cloud. *Proc. VLDB Endow.* 16, 12 (2023), 3488–3500.

[28] Hua Fan, Dachao Fu, Xu Wang, Jiachi Zhang, Chaoji Zuo, Zhengyi Wu, Miao Zhang, Kang Yuan, Xizi Ni, Huo Guocheng, Wenchao Zhou, Feifei Li, and Jingren Zhou. 2024. Towards Millions of Database Transmission Services in the Cloud. *Proc. VLDB Endow.* 17, 12 (2024), 4001–4013.

[29] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers* 21, 9 (1972), 948–960.

[30] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *SIGMOD*. 1917–1923.

[31] Martin Hilbert and Priscila López. 2011. The world's technological capacity to store, communicate, and compute information. *science* 332, 6025 (2011), 60–65.

[32] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[33] 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 (2020), 3072–3084.

[34] John F Hughes. 2014. *Computer graphics: principles and practice.*

[35] Adam Jacobs. 2009. The pathologies of big data. *Commun. ACM* 52, 8 (2009), 36–44.

[36] Maqbool Khan, Xiaotong Wu, Xiaolong Xu, and Wanchun Dou. 2017. Big data challenges and opportunities in the hype of Industry 4.0. In *ICC*. IEEE, 1–6.

[37] Alexandros Labrinidis and H. V. Jagadish. 2012. Challenges and Opportunities with Big Data. *Proc. VLDB Endow.* 5, 12 (2012), 2032–2033.

[38] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *nature* 521, 7553 (2015), 436–444.

[39] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanael Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, Ana Klimovic, Andy Slowey, and Antony I. T. Rowstron. 2017. Understanding Rack-Scale Disaggregated Storage. In *USENIX HotStorage.*

[40] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud Databases: New Techniques, Challenges, and Opportunities. *Proc. VLDB Endow.* 15, 12 (2022), 3758–3761.

[41] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *USENIX ATC*. 53–68.

[42] Peter M. Mell and Timothy Grance. 2011. The NIST Definition of Cloud Computing. *Special Publication (NIST SP)* 800-145 (2011).

[43] Michael P. Mesnier, Gregory R. Ganger, and Erik Riedel. 2003. Object-based storage. *IEEE Commun. Mag.* 41, 8 (2003), 84–90.

[44] Alexey Milovidov, Yakov Olkhovskiy, and Ivan Zhukov. 2021. ClickHouse: An Analytic DBMS for Interactive Applications. *PVLDB* 14, 12 (2021), 3235–3247.

[45] Lory D. Molesky and Krithi Ramamritham. 1995. Recovery Protocols for Shared Memory Database Systems. In *SIGMOD*. 11–22.

[46] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance.. In *CIDR*, Vol. 20. 29.

[47] Pedro Pedreira, Krishna Puttaswamy, Xiaoxuan Meng, Marios Kokkodis, Orri Erling, Nikhil Benesch, and Brian Nixon. 2023. Velox: Meta's Unified Execution Engine. In *SIGMOD*. 2221–2234.

[48] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, et al. 2022. Cloud-native transactions and analytics in singlestore. In *SIGMOD*. 2340–2352.

[49] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *SIGMOD*. 1981–1984.

[50] Mathijs Jeroen Scheepers. 2014. Virtualization and containerization of application infrastructure: A comparison. In *21st twente student conference on IT*, Vol. 21. 1–7.

[51] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow.* 17, 12 (2024), 3731–3744.

[52] Dharma Shukla, Shireesh Thota, Karthik Raman, Madhan Gajendran, Ankur Shah, Sergii Ziuzin, Krishnan Sundaram, Miguel Gonzalez Guajardo, Anna Wawrzyniak, Samer Boshra, Renato Ferreira, Mohamed Nassar, Michael Koltachev, Ji Huang, Sudipta Sengupta, Justin J. Levandoski, and David B. Lomet. 2015. Schema-Agnostic Indexing with Azure DocumentDB. *Proc. VLDB Endow.* 8, 12 (2015),

1668–1679.

[53] Michael Stonebraker. 1986. The Case for Shared Nothing. *IEEE Database Eng. Bull.* 9, 1 (1986), 4–9.

[54] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[55] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *SIGMOD*. 1493–1509.

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

[57] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD*. 789–796.

[58] Wei Wang, Meihui Zhang, Gang Chen, HV Jagadish, Beng Chin Ooi, and Kian-Lee Tan. 2016. Database meets deep learning: Challenges and opportunities. *SIGMOD Record* 45, 2 (2016), 17–22.

[59] Chuangxian Wei, Bin Wu, Sheng Wang, Renjie Lou, Chaoqun Zhan, Feifei Li, and Yuanzhe Cai. 2020. AnalyticDB-V: A Hybrid Analytical Engine Towards Query Fusion for Structured and Unstructured Data. *Proc. VLDB Endow.* 13, 12 (2020), 3152–3165.

[60] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (2022), 3385–3397.

[61] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.