Modernization of Databases in the Cloud Era: Building Databases that Run like Legos

Feifei Li
Alibaba Group
lifeifei@alibaba-inc.com

ABSTRACT

Utilizing cloud for common and critical computing infrastructures has already become the norm across the board. The rapid evolvement of the underlying cloud infrastructure and the revolutionary development of AI present both challenges and opportunities for building new database architectures and systems. It is crucial to modernize database systems in the cloud era, so that next generation cloud native databases may run like legos–they are adaptive, flexible, reliable, and smart towards dynamic workloads and varying requirements.

That said, we observe four critical trends and requirements for the modernization of cloud databases: embracing cloud-native architecture, full integration with cloud platform and orchestration, co-design for data fabric, and moving towards being AI augmented. Modernizing database systems by adopting these critical trends and addressing key challenges associated with them provide ample opportunities for data management communities from both academia and industry to explore. We will provide an in-depth case study of how we modernize *PolarDB* with respect to embracing these four trends in the cloud era. Our ultimate goal is to build databases that run just like playing with legos, so that a database system fits for rich and dynamic workloads and requirements in a self-adaptive, performant, easy-/intuitive-to use, reliable, and intelligent manner.

PVLDB Reference Format:
Feifei Li. Modernization of Databases in the Cloud Era: Building Databases that Run like Legos. PVLDB, 16(12): 4140 - 4151, 2023.
doi:10.14778/3611540.3611639

1 INTRODUCTION

Cloud has become the de-factor IT infrastructure standard for large and small enterprises across almost all industry sectors. With the increasing deployment of mission critical systems for businesses ranging from transaction processing to analytical processing on the cloud, fundamental changes are required for moving database systems to the cloud. In particular, we need to transform *cloud-hosted databases* to *cloud-native databases* so that database systems take the best advantages of the underlying cloud infrastructure. Meanwhile, with the rapid deployment of AI and big data technologies, database systems need to be more intelligent, so that they are easy and intuitive to use. In summary, database systems need to be modernized in the cloud-era, so that they are born to be both cloud native and AI native.

That said, we identify four critical trends and requirements for the modernization of cloud databases: *embracing cloud-native architecture, full integration with cloud platform and orchestration, co-design for data fabric*, and *moving towards being AI augmented.* Modernizing database systems by adopting these critical trends and addressing key challenges associated with them provide ample opportunities for data management communities from both academia and industry to explore. We will provide an in-depth case study of how we modernize *PolarDB* with respect to embracing these four trends in the cloud era.

In particular, major cloud service providers have deployed the latest networking infrastructure, memory/storage technologies, and interconnected heterogeneous computing devices in their IDCs, such as RDMA, persistent memories, CXL memories, cloud storage, FPGA, GPUs, DPAs such as bluefiled and more. Meanwhile, the orchestration of cloud resources has evolved quickly from hypervisor-based virtual machines to light-weight containers using kubernetes (and a mixture of the two). The first generation of cloud native database explores the decouple of computation and storage, so that computation nodes and storage nodes in a database system can independently scale. That is no longer sufficient, in order to fully explore the potential of what the underlying cloud infrastructure has to offer. The new generation of cloud native database systems needs to disaggregate storage, memory, and cpu cores, so that advanced features such as serverless, multi-master can be more naturally developed. At the same time, orchestration of (often, the computation nodes) container-based cloud native database instances becomes a critical need to achieve high availability, high elasticity and high resource utilization efficiency.

Furthermore, as more and more data are moving to the cloud and being generated in the cloud, business applications on the cloud increasingly demand agile development and deployment frameworks for their data processing needs, ranging from transaction processing to analytical processing. The concept of data-fabric becomes both popular and important, that is to enable the easy-sharing and the smooth-flow of data in-between different data processing engines and eco-systems without trouble. This requires the co-design of different cloud native database systems to achieve data-fabric, such as zero-ETL between OLTP and OLAP database systems, cloud-native HTAP, sharing of meta-data among different cloud native database systems, etc.

Lastly, with the eruptive development of ML and AI technologies, LLM (large language model) being a latest example, integrating AI technologies natively inside a database system will become a standard practice. In addition to having ML-based tuning, monitoring, and optimization for database systems (i.e., AIops for database

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611639

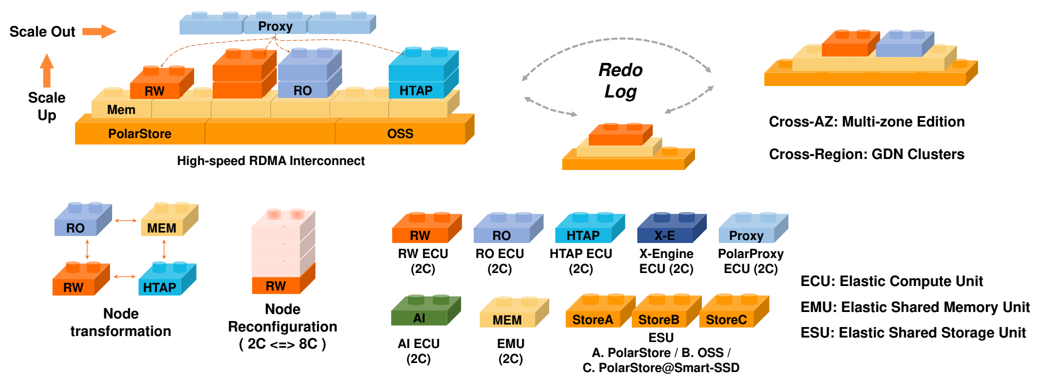

Scale Out
Scale Up
Proxy
RW
RO
HTAP
Mem
PolarStore
OSS
High-speed RDMA Interconnect
Redo Log
Cross-AZ: Multi-zone Edition
Cross-Region: GDN Clusters
RO
MEM
RW
HTAP
Node transformation
RW
Node Reconfiguration (2C <=> 8C)
RW ECU (2C)
RO ECU (2C)
HTAP ECU (2C)
X-E X-Engine ECU (2C)
Proxy PolarProxy ECU (2C)
AI ECU (2C)
MEM EMU (2C)
StoreA StoreB StoreC
ESU
A. PolarStore / B. OSS / C. PolarStore@Smart-SSD
ECU: Elastic Compute Unit
EMU: Elastic Shared Memory Unit
ESU: Elastic Shared Storage Unit

Figure 1: Building databases that run like legos.

systems), supporting AI inference inside a database system will also become widely available. Furthermore, vector database or vector engine will be a commodity in most database systems to store and processing high dimension vectors produced by the embedding process of various LLMs. ML and AI technologies will also change the way how we interact with a database system. NL2SQL (and NL2API) interface will witness wide adoption in BI and data science applications.

In this keynote, we will highlight the key challenges and development of modern cloud native databases using *PolarDB* as an example. Our ultimate goal is to build databases that run just like playing with legos as shown in Figure 1, so that a database system fits for rich and dynamic workloads and requirements in a self-adaptive, performant, easy-/intuitive-to use, reliable, and intelligent manner.

2 MODERNIZATION OF CLOUD DATABASES

2.1 Four Trends

As mentioned above, we have observed four critical trends and requirements for the modernization of cloud databases: embracing cloud-native architecture, full integration with cloud platform and orchestration, co-design for data fabric, and moving towards being AI augmented. We will first give an overview of *PolarDB* and then provide an in-depth study of *PolarDB* to demonstrate the development of modern cloud databases with respect to these four critical trends and requirements.

2.2 A Case Study: *PolarDB*

PolarDB [26] is a cloud native database system developed at Alibaba Cloud, and adopts a *shared storage* architecture. It is derived from the MySQL code base and uses PolarFS [11] as its underlying storage pool. It includes one primary read-write node (*i.e.*, RW node) and multiple read-only replicas (*i.e.*, RO nodes) in the compute node layer. Like a traditional database kernel, each RW or RO node contains a SQL processor, a transaction engine (like InnoDB [32], X-Engine [23]), and a buffer pool to serve queries and transactions. In addition, there are some stateless proxy nodes for the purpose of load balancing.

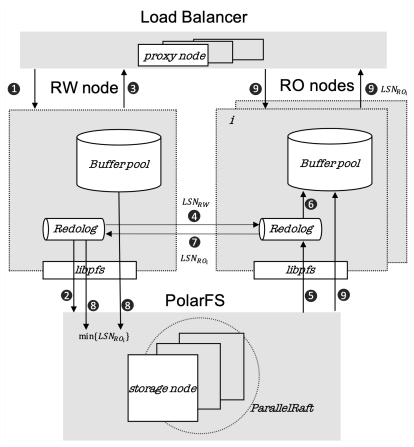
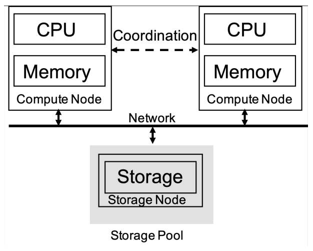


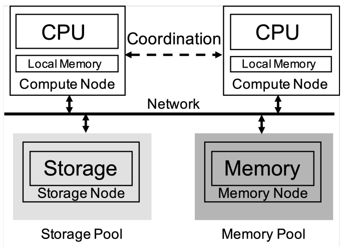
Figure 2: *PolarDB* Architecture.

PolarFS is a durable, atomic and scale-out distributed storage service. It provides virtual volumes that are partitioned into chunks of 10GB size, which are distributed into multiple storage nodes. A volume contains up to 10000 chunks, and can provide a maximum capacity of 100TB. These chunks are provisioned on demand, so that volume space can grow dynamically. Each chunk has three replicas, and linear serializablility is guaranteed through *Parallel Raft*, which is a consensus protocol derived from Raft.

The RW node and RO nodes synchronize memory status through redo logs, and coordinate consistency through log sequence number (LSN), which indicates an offset of redo log files (*e.g.*, in InnoDB). As shown in Figure 2, in a transaction ①, after RW finishes flushing all redo log records to PolarFS ②, the transaction can be committed ③. RW broadcasts messages that the redo log have been updated as well as the latest LSN lsn_{RW} to all RO nodes asynchronously ④. After the node RO_i receives the message from RW, it pulls updates of redo log from PolarFS ⑤, and applies them to the buffered page in buffer pool ⑥, so that RO_i keeps synchronized with RW. Then RO_i piggybacks the consumed redo log offset lsn_{RO_i} in the reply and sends it back to RW ⑦. RW can purge the redo log before the $min\{lsn_{RO_i}\}$ location, and flush the dirty pages elder than $min\{lsn_{RO_i}\}$ to PolarFS in the background ⑧. While RO_i can serve



(a) *shared storage architecture*



(b) *disaggregation architecture*

Figure 3: *Architectures for resource disaggregation.*

read transactions using the snapshot isolation with version before lsn_{RO_i} ⑨. Some RO nodes may fall behind because of high CPU utilization or network congestion. Say there is a certain node RO_k, whose LSN lsn_{RO_k} is much lower than that of RW lsn_{RW} (the lag is larger than one million). Such node RO_k will be detected and kicked out of the cluster to avoid slowing down RW through dirty page flushes.

Those stateless proxy nodes provide transparent load balancing service for separating read and write traffics, *i.e.*, distributing read requests to RO nodes and forwarding write requests to RW.

3 CLOUD-NATIVE ARCHITECTURE

This section introduces *PolarDB*'s capabilities that embrace the advantages of cloud-native architecture. They have significantly strengthened the system's flexibility (*e.g.*, via resource disaggregation), efficiency (*e.g.*, via multi-master replication), and consistency (*e.g.*, via strong-consistent reads).

3.1 Resource Disaggregation

3.1.1 Background. There are three typical architectures for cloud databases: 1) *monolithic machine*; 2) *virtual machine with remote disk*; and 3) *shared storage* as shown in Figure 3(a). The last two can be referred as the *decouple of computation and storage.* Though these architectures have been widely used, they all suffer from challenges caused by resource coupling.

Under the *monolithic machine* architecture, all resources (*e.g.*, CPU, memory and storage) are tightly coupled. Different resources allocated on a physical machine are difficult to sustain at a high utilization rate, and hence they are prone to fragmentation. In addition, a system with tightly coupled resources has the problem of fate sharing, *i.e.*, the failure of one resource will cause the failure of other resources, leading to longer system recovery time. With the *decouple of computation and storage* architecture, DBaaS (DB as a service) can independently improve the resource utilization of the storage pool. The *shared storage* subtype further reduces storage costs — the primary and read replicas can attach and share the same storage. Read replicas help to serve high-volume read traffic and offload analytical queries from the primary. However, in all these architectures, problems like bin-packing of CPU and memory, lacking of flexible and scalable memory resources, remain unsolved. Furthermore, each read replica keeps a redundant in-memory data copy, leading to high memory costs.

3.1.2 PolarDB serverless. To address above issues, we propose a novel cloud database paradigm called *disaggregation* architecture [13] as shown in Figure 3(b). It goes one step further than the *shared storage* architecture. The disaggregation architecture runs in the disaggregated data centers (DDC), in which CPU, memory and storage resources are no longer tightly coupled as in a monolithic machine. Resources are located in different nodes connected through high-speed network. As a result, each resource type can improve its utilization rate and expand its volume independently. This also eliminates fate sharing, *i.e.*, allowing each resource be recovered from failure and upgraded independently. Moreover, data pages in the remote memory pool can be shared among multiple database processes, analogous to the storage pool being shared in *shared storage* architecture. Adding a read replica no longer increases the cost of memory resources, except for consuming a small piece of local memory.

Following this *disaggregation* architecture, we build *PolarDB* Serverless [13]. It introduces a multi-tenant scale-out memory pool, including page allocation and life cycle management. Our first challenge is to ensure that the system executes transactions *correctly* after adding remote memory to the system. For example, read after write should not miss any updates even across nodes. We solve it using cache invalidation. When RW is splitting or merging a B+Tree index, other RO nodes should not see an inconsistent B-tree structure in the middle. We protect it with global page latches. When a RO node performs read-only transactions, it must avoid reading anything written by uncommitted transactions. We achieve it through the synchronization of read views between database nodes.

Besides, the evolution of the *disaggregation* architecture could have a negative impact on the database performance. It is because the data is likely to be accessed from the remote, which introduces significant network latency. Our second challenge is to execute transactions *efficiently*. We exploit RDMA optimization extensively, especially one-sided RDMA verbs, including using RDMA CAS [42] to optimize the acquisition of global latches. In order to improve concurrency, both RW and RO nodes use optimistic locking techniques to avoid unnecessary global latches. On the storage side, page materialization offloading allows dirty pages to be evicted from remote memory without flushing them to the storage, while index-aware prefetching improves query performance.

Finally, the *disaggregation* architecture complicates the system and hence increases the variety and probability of system failures. As a cloud database service, our third challenge is to build a *reliable* system. We derive our strategies to handle single-node crashes of different node types, which guarantee that there is no single-point failure in the system. Because the states in memory and storage are decoupled from the database node, crash recovery time of the RW node becomes 5.3 times faster [13] than that in the *monolithic machine* architecture.

3.2 Multi-Master Replication

3.2.1 Background. Recall that *PolarDB* consists of one primary (RW) node to process the read/write requests and one or more secondary (RO) nodes to handle read-only requests. However, in write-heavy workloads, the single primary node will become the

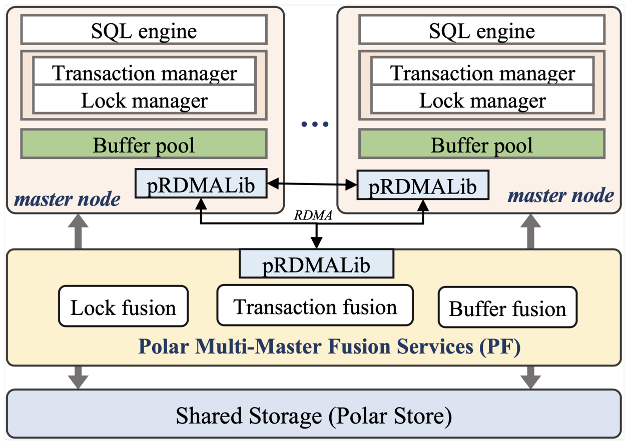


Figure 4: Architecture of *PolarDB*-MM.

bottleneck. We have to scale up the primary node when the write pressure becomes heavy, but it is still restricted by the physical machine's specification. When the primary node fails, one of the secondary nodes will be promoted to the new secondary node. However, this kind of high-availability still induces a brief downtime during failover [1]. Thus, multi-master cloud-native databases are highly required to support high scalability and availability.

3.2.2 *PolarDB-MM.* We design and implement *PolarDB*-MM (*PolarDB* Multi-Master) on top of *PolarDB*'s shared storage architecture. It still uses PolarFS [11] as the shared storage, but multiple master nodes are connected with the shared storage. All master nodes have equal access to the shared storage. What's more, we use modern RDMA network to speed up communication among different masters.

Figure 4 shows the overview architecture. All master nodes share the same data and have equal access to the shared data, and they could process read/write requests simultaneously. The core component is the *Polar Multi-Master Fusion Services* (PF) that serves the master nodes to achieve concurrent transaction execution. The master nodes and PF are connected via the high-speed RDMA network. Our highly-optimized RDMA library supports ultra-low-latency communication between masters and PF. PF only serves master nodes, having no connection to the storage. All the I/Os are issued on the master nodes.

PF has three components, *i.e.*, *buffer fusion*, *transaction fusion* and *lock fusion*. Buffer fusion is designed to achieve buffer coherency between different master nodes. In *PolarDB*-MM, each master node has its own buffer pool and PF maintains a global buffer pool in its buffer fusion module. Since all master nodes have equal access to the shared storage, each data page can be read/written by any master. So we design the distributed page locking scheme to control the inter-master concurrent page data access. A master has to require a shared/exclusive lock to read/write a page from the global buffer pool. Once a master updates a page, it will invalidate that page on other masters' local buffer pools before releasing the page lock. Buffer fusion achieves the page's consistent state in the multi-master cluster.

Based on the consistent page state, transaction fusion further supports the concurrent transaction execution on multiple masters, while guaranteeing the transaction's ACID. Transaction fusion provides a centralized Timestamp Oracle (TSO) that monitors the commits of all transactions. Each transaction has to request a global commit timestamp (CTS) from the TSO before committing. The CTS reflects the transaction's ordering. *PolarDB*-MM also supports MVCC to achieve high-throughput and lock-free snapshot reads. Transaction fusion provides a novel transaction system to speed up the transaction's visibility decision.

Lock fusion supports the row-level lock for the concurrent transaction's execution on different master nodes. To support efficient locking on different nodes, we store the lock information with the row data. This could save a lot of overhead to maintain the lock information in a global data structure.

Our evaluation shows that *PolarDB*-MM has linear scalability when different masters access different parts of the dataset, and each node's throughput only drops by 15%-30% when they uniformly access all the data in SysBench's read-write workload.

3.3 Strong-consistent Reads

3.3.1 *Background.* In *PolarDB*, to keep an RO node's buffered data up-to-date, the RW node generates the corresponding log for each update and ships the log to RO nodes. RO nodes apply the log to update their buffered data. Since the log application process is asynchronous, an RO node may be unable to return the latest updates that have already taken place on the RW node and consequently could return "stale" data. Many cloud-native databases claim that RO nodes could improve read performance. However, for the reason outlined above, the service on an RO node can only serve applications that does not require read-after-write consistency.

However, the strongly consistent read (*i.e.*, a read request always sees the latest committed updates that happen before it, aka *the strict consistency model* [48]) is an essential need in many applications [2]. For instance, in Alibaba's e-commerce applications, if strong consistency cannot be guaranteed, the customer who has placed an order may soon finds that the order does not exist or is shown to be unpaid after payment. Such need for strongly consistent reads also appear in scenarios where databases are used to support interactions among microservices [25]. These microservices usually share the same databases and have some dependencies at the application level. It relies the database to provide strong consistency to ensure the interactions proceed as expected. At Alibaba Cloud, we also receive many requests from users to provide such strongly consistent reads, such as insurance companies and financial institutes.

3.3.2 *PolarDB-SCC.* To support strong consistency, applications have to send all read requests to the RW node. Consequently, we cannot improve the read throughput by adding more RO nodes and the RW node could quickly become the bottleneck. This dramatically limits the system's ability to process read-dominant workloads [8, 15, 43]. To support the RO node's auto-scaling-out, a unified endpoint is required for users. The strongly consistent read must be guaranteed by this endpoint to ensure that the writes on this endpoint must be immediately visible to the following reads. Therefore, it's imperative to have a new system design to ensure strongly consistent reads on RO nodes in a cloud-native database

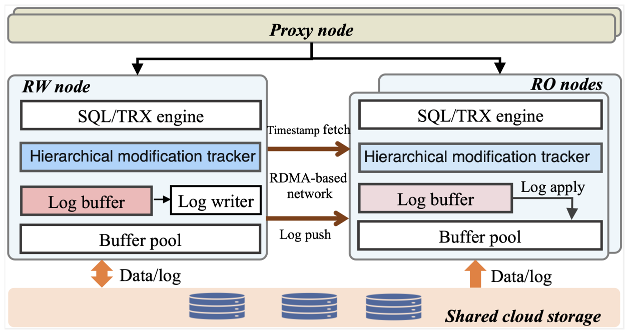


Figure 5: Architecture of *PolarDB*-SCC.

cluster to improve system performance and make the system really scalable.

To enable low-latency strongly-consistent reads on RO nodes, we build *PolarDB*-SCC [49] (*PolarDB* Strongly Consistent Cluster), which is designed with a read-wait policy. It aims to provide a low-latency, strongly consistent cloud-native database cluster, in which the RO nodes could always return the latest updates that are committed ahead of the request's/transaction's start timestamp. This enables the system to distribute read requests to the RO nodes and split the read/write requests while ensuring strong consistency. As a result, the cluster can provide a unified, strongly consistent endpoint for applications (*e.g.*, via a proxy), and adjust the number of RO nodes on-demand elastically. Resource utilization on RO nodes is also improved, rather than deploying RO nodes only for handling failover. The main challenge is to keep the in-memory data consistent between the RW and RO nodes while ensuring low latency. The key idea of *PolarDB*-SCC is to eliminate unnecessary waits and reduce the necessary wait time on the RO node.

Figure 5 shows the overall architecture. The core components are the hierarchical modification tracker, Linear Lamport timestamp, and the RDMA-based log shipment protocol. The hierarchical modification tracker maintains the RW node's modification at three levels: the global level maintains the whole database's latest modification timestamp; and table/page levels record some tables'/pages' newest modification timestamps. To perform a strongly consistent read on the RO node, it will first check the RW node's global level timestamp, then the table and page level timestamps. Once one level is satisfied, it will directly process the request and will not check the next one. It only needs to wait for the log application on the requested pages when the last level (page level) is unsatisfied.

Since the latest modification timestamps are maintained on the RW node, the RO node has to fetch it from the RW node for each request. Although the RDMA network is fast, the overhead is still significant if there is a heavy load on the RO node. To overcome the overhead on the timestamp fetching, we propose the Linear Lamport timestamp. Based on it, the RO node can store the timestamp locally after fetching it from the RW node. Any request arriving at the RO node earlier than TS_{ro} can directly use the locally stored timestamp instead of fetching a new one from the RW node. This can save many fetching requests when the load is heavy on RO nodes.

At last, to further minimize the network overhead, we adopt the one-sided RDMA for the log shipment and timestamp fetching. We propose a one-sided RDMA-based log shipment protocol to write the RW node's log to the RO nodes. The one-sided RDMA also saves a lot of CPU cycles during remote writing.

4 CLOUD PLATFORM INTEGRATION AND ORCHESTRATION

While cloud-native databases are becoming an inexorable trend in the database industry, its unique advantages root from the close integration and co-design with the underlying cloud infrastructure. In this section, we introduce two representative examples that reflect this concept. We present our best practice in resource scheduling of a unified resource pool in Section 4.1, and the adoption of computational storage devices that enable significant performance improvement in Section 4.2.

4.1 Unified Resource Pool

With the rising popularity of container infrastructures (such as Kubernetes), there has been a trend to host instances of cloud-native databases within containers, benefiting from their strong support for orchestration and migration. This helps cloud vendors achieve high availability, high elasticity and high resource utilization efficiency. A cluster management system is employed to manage the jobs or tasks (i.e., database instances) running on the cluster of machines (nodes). At the center of a cluster management system is a resource scheduler which dictates when and how cloud resources are allocated to different jobs.

4.1.1 Balance Allocation Rate and Availability. Typically, cloud vendors use two critical metrics to assess whether a resource scheduler is running "well": 1) *resource allocation rate* refers to the proportion of allocated resources (out of the total cluster resources) that have been allocated by the scheduler to a job; 2) *resource availability* refers to the proportion of resource requests (from a job) that can be fulfilled within a given period of time. Naturally, cloud vendors aim for high resource allocation rate (directly leading to lower operation cost) and high resource availability (directly leading to better customer experience). However, it is intuitively difficult to simultaneously maximize these two metrics. In fact, it depends on the application scenario and requires balancing the optimization of the resource allocation rate and resource availability.

4.1.2 Eigen Resource Scheduler. We propose a resource scheduling strategy that improves resource allocation rate without hurting resource availability. We adopt a cascading resource flow model that divide nodes in a cluster into three types: non-empty online nodes, empty online nodes, and offline nodes. In order to simultaneously maximize resource allocation rate and resource availability, we present Eigen [28], an *end-to-end resource optimizer* that optimizes the resources in the resource flow simultaneously. We divide machines of each node pool into three layers: online layer (green box), warm layer (orange box), and cold layer (blue box). Figure 6 depicts the resource flows that show how machines move between layers and node pools. We describe the optimization problems at each layer as follows:

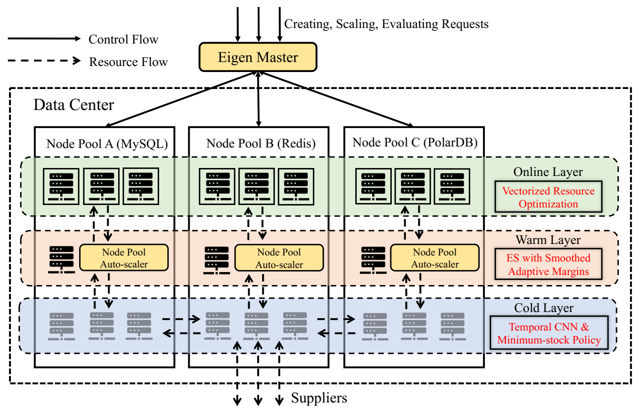


Figure 6: **EIGEN's hierarchical resource management system.**

Online layer consists of non-empty machines (online machines). In this layer, the optimization problem is to allocate database instances with heterogeneous resource requests on as few as possible online machines. We design Vectorized Resource Optimization (VRO), which consists of an online version and an offline version. The online version of VRO is implemented in *Scheduler* to schedule resource allocations for online requests. To support regional scheduling on large-scale clusters of up to 100K machines, the computation of scheduling should execute as efficiently as possible. The offline version of VRO is implemented in *Rebalancer*, which periodically rebalances the cluster through pod moves (i.e., migrations of database instances). In addition to consolidating clusters, the offline version of VRO focuses on reducing the number of migrations for lower rebalance costs.

Warm layer consists of empty machines (warm machines) which work as "buffers" to support high resource availability. The optimization problem in this layer is to evaluate the minimum of resources which will not cause delayed requests in short-term time periods (e.g., ten seconds, one minute, ten minutes). We design Exponential Smoothing (ES) with smoothed adaptive margins, a resource reservation algorithm that predicts short-term resource usage with a smoothed adaptive margin. It is implemented in *Node Pool Auto-scaler*, and automatically scales up/down warm machines.

Cold layer consists of offline machines (cold machines). In this layer, the optimization problem is to evaluate the minimum of cold machines which will not cause failed requests in a long-term time period (e.g., one week, three weeks, one month). We train and deploy probabilistic time-series forecasting models on *Node Pool Auto-scaler* to predict long-term daily resource consumption (i.e., a daily difference of allocated resources). Based on the predictions, we design a Minimum-stock Policy that suggests on adding or removing cold machines.

4.2 Co-design of Software and Hardware

To best serve OLTP workloads, cloud-native relational databases, such as PolarDB, typically employ the row-store model. A viable option for these database to better serve analytical workloads is to offload data-access-intensive tasks (in particular table scan) from database nodes to storage nodes. In spite of the simple concept, its practical implementation in the context of cloud-native databases is particularly non-trivial, and requires careful co-design of software and hardware. On one hand, each storage node must be equipped with sufficient data processing power to handle table scan tasks. On the other hand, to maintain the cost effectiveness of cloud-native databases, we cannot significantly (or even modestly) increase the cost of storage nodes. By complementing CPUs with special-purpose hardware (e.g., GPU and FPGA), heterogeneous computing architecture appears to be an appealing option to address this data processing power vs. cost dilemma. Under this framework, each data storage device becomes a computational storage drive that can carry out table scan on the I/O path. However, its practically viable implementation is challenging, mainly due to the difficulty of addressing two challenges:

- **Support pushdown across the entire software hierarchy**: Table scan pushdown is initiated by PolarDB storage engine that accesses data by specifying the offsets in files, while table scan is physically served by computational storage drive that operates as a raw block device and manages data with LBA (logical block address). The entire storage I/O stack sits in between the PolarDB storage engine and the computational storage drive. Hence, we have to cohesively enhance/modify the entire software/driver stack in order to create a path for table scan pushdown.
- **Implement low-cost computational storage drive**: Although the FPGA-based design approach can significantly reduce the development cost, FPGA tends to be expensive. Moreover, since FPGA typically operates at only 200-300MHz (in contrast to 2-4GHz CPU clock frequency), we have to employ a large degree of circuit-level implementation parallelism (hence more silicon resource) in order to achieve sufficiently high performance. Therefore, we must develop solutions to enable the use of low-cost FPGA chip in our implementation.

To address these two challenges, PolarDB adopts a set of software/hardware techniques [10]: To reduce the product development cycle and meanwhile ensure cost effectiveness, computational storage drives use an FPGA-centric host-managed architecture. Inside each computational storage drive, a single low-cost Xilinx FPGA chip handles both flash memory control and table scan. With highly optimized software and hardware design, each computational storage drive can support high-throughput (i.e., over 2GB/s) table scan on compressed data and meanwhile achieve storage I/O performance comparable to leading-edge NVMe SSDs.

In addition to offloading scan operations to the computational storage devices, PolarDB has further leveraged smartSSD to performance transparent data compression. The compression is performed by the FPGA (or ASIC) colocated with SSDs, delivering over 60% reduction in storage (compared to uncompressed data) with no negative impact on throughput and little increase in the cost of SSDs. Co-design of software and hardware extends beyond storage devices; for example, the emergence of new technologies such as software defined network (SDN) and Compute Express Link (CXL) opens new directions that explores in-network computation and memory pooling and sharing.

5 CO-DESIGN FOR DATA FABRIC

Another trend that we have witnessed is that the line between different types of database engines start to blur: there is a growing need for a database to provide sufficient support for a wide variety of different workloads (notably, *both* transactional processing and analytical processing), especially in the fields of business intelligence [44], social media [7, 31], fraud detection [9], and marketing [19, 52]. To provide such capability, traditional solutions often deploy data and application logic into multiple databases specialized in processing different types of queries, and rely on data synchronization techniques (such as ETL) for consistencies among them. Such solutions are costly, as it negatively impacts the OLTP performance, and introduces a time-consuming data synchronization process, which further leads to delays or even inconsistencies between the data maintained at the TP/AP databases. Additionally, users are often provided with isolated portals or interfaces for access each of the different databases. In practice, these issues lead to sub-optimal user experience.

The concept of data fabric emerges in recent years, promoting easy-sharing and smooth-flow of data in-between different data processing engines and even different eco-systems. In this section, we discuss the designs that enables unified database interface through meta-data sharing among engines (Section 5.1), and allows smooth data flow in-between different database engines through zero-ETL and cloud-native HTAP (Section 5.2).

5.1 Unified Database Interface

The wide variety of database engines allows customers to have different choices for their data processing needs, however, at the same time, it poses challenges to the customers: they need to make "wise" decisions in terms of which engine fits their needs the best, and how the data should be stored (e.g., within the same engine or across multiple engines). It calls for a unified database interface to simplify this process and alleviate the burden.

- **G#1: Transparent Query Execution.** To serve mixed workloads in a single database, database users should not be required to understand the working logic of the database, nor should they identify query types manually. That is, users should not perceive multiple *isolated* systems (e.g., engines, indexes, interfaces, etc.) for workloads with different characteristics.

To achieve this, it requires cross-engine sharing of metadata, such that the database system can make decisions from a holistic view. We adopt a unified management of metadata, including table schema and statistics, data lineage, database topology, and execution history. Based on these information, the database system's optimization and fabric layer can generate and coordinate, through a rule-based or cost-based process, executions in different engines.

5.2 Data Flow in-between DB Engines

With the unified execution, the data engine responsible for handling writes (e.g., insertion, deletion and update) might not be the one that is best fit for reads. Particularly, OLTP engines are often optimized for write performance, while OLAP engines are optimized for processing complex analytical queries. The data flow in-between DB engines ideals should satisfy the following properties:

- **G#2: Advanced OLAP Performance.** As a major goal of any HTAP database, its OLAP performance (e.g., execution latency) should be comparable to typical databases specialized in processing OLAP queries (typically through the introduction of columnar data storage).

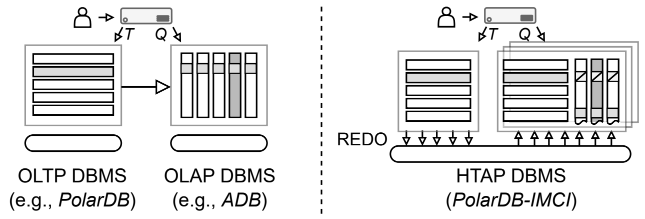


Figure 7: Architecture of HTAP databases.

- **G#3: Minimal Perturbation on OLTP Workloads.** While the performance of OLAP queries is significantly improved, it should have a minimal negative impact on the performance of OLTP queries. In fact, as we have practically validated in real application scenarios, OLTP queries are usually more mission-critical and are more sensitive to performance degradation. This requires effective resource isolation for OLTP and OLAP queries.
- **G#4: High Data Freshness.** High data freshness is an important property of HTAP databases, which is a distinguishing advantage compared to the traditional Extract-Transform-Load (ETL) method. Conventionally, visibility delay is used as a freshness score for a query: visibility delay is the time interval during which updates to the database can be visible to OLAP queries.

Figure 7 summarizes two approaches that are designed with these goals in mind: the system could adopt a zero-ETL approach (shown on the left), where the data synchronization, typically using logical logs, is co-designed with the source and destination engines for achieving goal G#2-G#4; one could alternatively adopt a cloud-native HTAP approach (shown on the right), where the synchronization is done through the shared storage, using physical logs, and an in-memory column index (IMCI) is built along side the row-oriented data for facilitate the processing of analytical queries. The zero-ETL approach is more general while the IMCI approach allows for minimal visibility delay (i.e., highest data freshness).

5.2.1 Zero-ETL between OLTP and OLAP. The key design of the zero-ETL approach focuses on reducing the resource as well as the time needed for transfering data from OLTP to OLAP. Crucially, to address the discrepancy between the write performance of OLTP and OLAP engines. We adopt a *Delta + Main* design:

For general stream writing scenarios, data is written to a write-optimized store (i.e., the *delta* store), which typically is in the form of an LSM-tree. The goal is to fully utilize the capabilities of row storage and make up for the performance shortcomings of columnar storage. Then a compaction process running at background will automatically sort, merge, and eventually write the data to a read-optimized storage (i.e., the *main* store): Data is first sorted by a predefined sort key. Once sorted, the data is easier to prune during scanning, reducing I/O overhead and improving scanning performance. Then the database merges these write-friendly delta data into read-friendly columnar storage, and promptly reclaims the storage space.

5.2.2 Cloud-native HTAP. Figure 8 shows the architecture of PolarDB-IMCI, a cloud native HTAP designed and operated by Alibaba Cloud [45]. It adopts PolarFS [12] as its storage layer, and a computation layer that contains multiple computation nodes, including a primary node for read/write requests (RW node), several nodes for

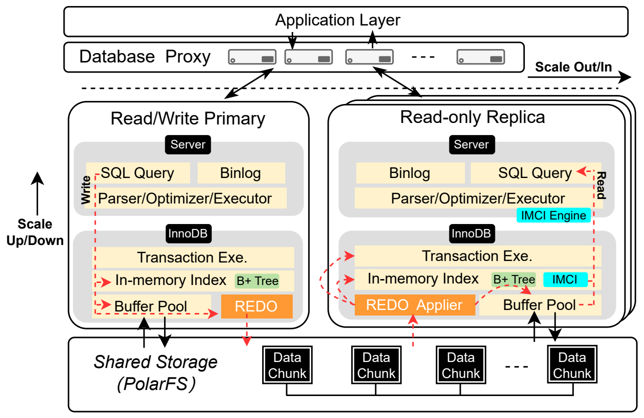


Figure 8: Architecture of PolarDB-IMCI.

read-only requests (RO nodes), and several stateless proxy nodes for load balancing. To speed up analytical queries, PolarDB-IMCI supports building in-memory column indexes on the row store of RO nodes. Column indexes store data in insertion order and perform out-place writes for efficient updates. The insertion order means a row in column indexes that can be quickly located by its Row-ID (RID) rather than its primary key (PK). To support PK-based point lookups, PolarDB-IMCI implements a RID locator (i.e., a two-layer LSM tree) for PK-RID mapping.

PolarDB-IMCI uses an asynchronous replication framework for synchronization between RO and RW. That is, updates to RO nodes are not included in the transaction commit path of the RW to avoid the impact on the RW node. To enhance data freshness on RO nodes, PolarDB-IMCI uses two optimizations on the log applying, the commit-ahead log shipping, and the conflict-free parallel log replay algorithm. RO nodes are synchronized by REDO logs of the row store, which causes very low perturbation on OLTP than the approaches that uses logical logs (such as Binlogs). Note that it's nontrivial to apply physical logs into column indexes as the data format of the row store and column index is heterogeneous.

Inside each RO node, PolarDB-IMCI uses two mutually symbiotic execution engines: PolarDB's regular row-based execution engine to serve OLTP queries, and a new column-based batch mode execution engine for the efficient running of analytical queries. The batch mode execution engine draws on the techniques used by columnar databases to handle analytical queries, including a pipeline execution model, parallel operators, and a vectorized expression evaluation framework. The regular row-based execution engine with augmented optimizations can undertake the column engine's incompatible queries or point queries.

6 AI AUGMENTED DATABASES

With the rapid progress in artificial intelligence (AI) technologies, PolarDB takes a proactive approach to integrate AI features into database systems, so as to be easier to use, more efficient to operate, and exhibit certain intelligence in deriving valuable insights from data. To this end, we have put into production a number of AI-augmented functionalities driven by customer needs. The most relevant features are two: an enhanced natural language interface that can convert questions into SQL statements [20], and a diagnosis system to pinpoint root causes for performance issues[22, 29]. These new features can enhance database systems by automating data management processes, improving user experience, and enabling intelligent decision-making in DevOps.

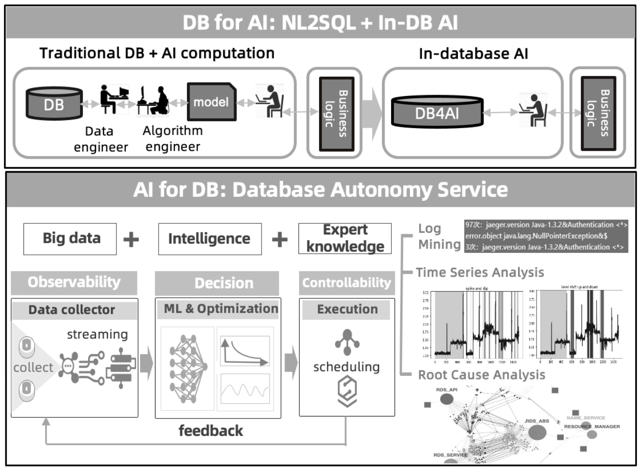


Figure 9: AI-augmented databases

6.1 NL2SQL

Natural language to SQL (NL2SQL) techniques provide a convenient interface to access databases, especially for non-expert users, to conduct various data analytics. With prior knowledge on the database schema information, PolarDB can automatically translate the natural language question into the corresponding SQL query. To provide high-quality translation, there are three challenges: (1) what tables and columns should be used in the query; (2) what is the correct query structure; and (3) how to fill in query details and the literal in the query.

Recently, the research on natural language models have made significant breakthroughs. Deep models based on transformer architectures have demonstrated great performance due to its excellent in-context learning capabilities. Interestingly, SQL languages are traditionally investigated in the framework of context-free grammars. The former captures complex meanings, and generalizes well, but may result in queries with syntactic or semantic errors. The latter can define high level structures with rigorous and accurate grammars, but may miss the global picture of the whole sentence. We introduce the NL2SQL module for PolarDB, by bridging the gap between the two and design a new framework to significantly improve both accuracy and runtime. In particular, we develop a novel sketch [20], which constructs a template with slots that initially serve as placeholders, and tightly integrates with a deep learning model to fill in these slots with meaningful contents based on the database schema.

Compared with the widely used sequence-to-sequence approaches, our sketch-based method does not need to generate keywords which are boilerplates in the template, and can achieve better accuracy and run much faster. Compared with the existing sketch-based approaches, our method is more general and versatile, and can leverage the values already filled in on certain slots to derive the rest ones for improved performance. In addition, we also leverages database domain knowledge, by introducing a post-processing module. It checks the initially generated SQL queries by applying rules to identify and correct semantic errors. This technique significantly improves the NL2SQL accuracy. Extensive evaluations on both single-domain and cross-domain benchmarks demonstrate that our approach can achieve great accuracy and throughput.

6.2 In-DB AI

In-DB AI is to reduce the unnecessary data movement between databases and AI computation engines. It can shorten the development cycle and lower the operational cost by integrating AI capabilities through SQL queries natively. We build an in-database inference framework that generates database loadable functions from already developed AI models. After registering the AI models, user could directly write SQLs to invoke model inference coherently into the query flow. To flexibly support models with different sizes and platforms, our framework considers three scenarios. First, we provide pre-installed model functions. Second, user can upload a trained model, which automatically generates static objects for shared files inside the database instances. We support models from Tensorflow, PyTorch and libraries such as XGBoost. Third, for large models, e.g., LLMs (Large Language Models), our framework generates a hook function to call external model inference service. For all of the above scenarios, users can write native SQL statements to augment data queries by AI models.

6.3 AIOps for Cloud Native Database Systems

PolarDB architecture follows a decoupling design principle, whose individual components, like any other distributed systems, may contain inevitable faults or failures, and are organized together through highly efficient message passing over RDMA networks. To make it robust and easier to manage, we design a new root cause diagnosis facility for its DevOps system[22, 29]. It is based on an observation that anomalies, or general variations, require a quantitative characterization of the influences from individual components on the end-to-end performance metrics.

On a causal graph that represents the complex dependencies between the system components, the scatteredly detected anomalies, even when they look similar, could have different implications with contrastive remedial actions. Though various heuristic methods have been successfully applied for certain cases in practice, the complexity constantly imposes various challenges for pinpointing the root causes. Various heterogeneous components are connected on a (non-strict) causal/relationship graph. Most existing works focus on predicting the performance issues (or more rigorously, counterfactuals) when a given set of components/factors change to a hypothetical operating state. Our work complements by quantifying the influences of the variations from the different factors using Shapley value, which is a concept in game theory to fairly allocate the influences of the factors. A factor's influence is measured by the average performance difference before and after adding this factor to a random set of existing factors. For example, one can study the difference of the end-to-end latency by adding an issue of high CPU usage. Our system can automatically drill down to the combinations of attributes where anomalies occur and evaluate the impact of each attribute values. This system has been successfully deployed with more than 10,000 operations for 86 services on 2,546 machines. It has significantly improved the DevOps efficiency, and greatly reduced the system failure rates.

7 OUTLOOK AND FUTURE TRENDS

7.1 LLM and Vector DB

Large language models (LLM) and vector databases, combined together, bring exciting new opportunities in various fields, thanks to the progresses in natural language processing and information retrieval. These technologies have significantly improved the way we interact with computers and access information.

Large language models, such as ChatGPT and Alibaba Tong Yi, are capable of generating human-like text and understanding the context and meaning of words and sentences. They can be used for a wide range of applications, including language translation, text summarization, and creative writing. Vector databases can store and retrieve data by representing them as vectors. It can compare and search for similar items based on their inherent characteristics. The combination of large language models and vector databases can enable more accurate and natural text generation with domain knowledge, as well as efficient and personalized data retrieval.

Vector database provides new dimensions in forming queries that enable more intelligence. In certain applications both unstructured and structured data shall be jointly queried. To address this challenge due to hybrid queries, we design and implement AnalyticDB-V (ADBV) [47] that manages feature vectors and structured data cooperatively. In order to improve the accuracy of querying massive data in vector forms, we design an index using Voronoi Graph Product Quantization, which could efficiently narrow down the search scope for fast indexing. In addition, it is wrapped as physical operators that can rely on the query optimizer to efficiently and effectively process hybrid queries.

7.2 Data Security

7.2.1 Background. During the past decade, practical techniques for database security have not witnessed significant advances, where access control, file encryption, database audit have long become de-facto standards [16, 17] that protect the database from unexpected accesses and external attacks. In conventional settings, a database system shall run in a private domain, and the system owners (as well as administrators) inherently have full access to the data inside. However, recent trends have overturned the assumption and brought new security issues. The first trend is cloud computing. Cloud systems, as an outsourced infrastructure, break the private domain assumption. and hence they might be potentially compromised by insiders (*e.g.*, co-tenants or rogue staffs). The second trend is that the data-centric revolution has complicated the data management in applications. More specifically, the data needs to flow between different processing components, each of which is probably controlled by a different entity (*e.g.*, internal subdivisions, business partners, and independent software vendors). Once the data flow into others' subsystems and databases, it is no longer under the control of the original data owner, leading to a contradiction between the utilization and the ownership of data. The data ownership here involves many aspects of data security

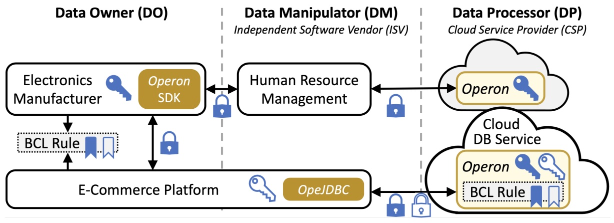


Figure 10: Illustration of supporting the OPDB with *Operon*. Data owners use their keys (differently colored) to protect data against any application processes or databases. It only performs permitted operations (*i.e.*, by BCL) within TEE.

issues, such as the confidentiality of users' sensitive data, and the authenticity of user's query results.

7.2.2 Encrypted database. Many encrypted database systems have been built by both academia and industry to protect the confidentiality of sensitive data in outsourced databases. They either exploit special cryptographic primitives [14, 21, 33] to support data manipulation directly over ciphertext [35, 37, 40] or use trusted execution environments (TEE) [18, 24] to operate on sensitive data in an isolated *enclave* inaccessible from the rest of the host [3, 5, 38, 39, 41]. However, existing solutions assume that the authorized endpoint directly interacting with the encrypted database is trusted and can touch sensitive data, which is hard to achieve in practice. Hence, we propose a new paradigm for the encrypted database, namely *ownership-preserving database* (OPDB) [46], with which the data is not even revealed to any subsystems and hence the data owner exclusively governs data accessibility. In a nutshell, all sensitive data remains in ciphertext wherever it appears (*e.g.*, in the memory of application/database server processes or on the disk) and only the data owner can decrypt the data. When an entity (*e.g.*, a business partner) needs to process or utilize the sensitive data in its business logic, the data owner only needs to grant it access to necessary operations that are adequate to complete the task using cipher processing capability in an OPDB instance. With OPDB, sensitive data can be securely passed and processed across different entities' subsystems and databases, which significantly reduces the risk of data abuses and leakages throughout the entire process.

Following the OPDB paradigm, we build *Operon* [46] as its first implementation, utilizing TEE to re-establish the private domain assumption. *Operon* introduces a *behavior control list* (BCL), which extends conventional system-oriented resource access control with the control of data operation behaviors, decoupling the data ownership away from the system ownership. *Operon* is the first database framework with which the data owner exclusively controls the accessibility and behaviors of sensitive data, even when the data has passed through many entities' untrusted subsystems. Figure 10 shows an example of supporting OPDB with *Operon*. It adopts a modular architecture and acts as a feature enhancement to existing database systems. We have successfully integrated it with different TEEs (*i.e.*, Intel SGX [18] and an FPGA-based implementation), as well as various database services on Alibaba Cloud (*i.e.*, *PolarDB* and RDS). In addition, since real-world applications always involve rich database functionalities (*e.g.*, mix-typed expression, connection pool, client driver) beyond basic operational primitives, we should also provide corresponding functionalities while preserving data ownership. *Operon* uses a set of server-side and client-side co-designs to solve this problem.

7.2.3 Verifiable database. Currently, data integrity largely relies on the customer's trust that the cloud service provider has faithfully maintained the data (and computed results) outsourced to them. It remains challenging for the customer to *verify* whether the data or computation results retrieved from the cloud are correct; from the cloud service provider's perspective, such capability of *proving* to the client that its data is correctly handled is also a highly desirable security feature, encouraging hesitant clients to adopt cloud-centric solutions. Therefore, *verifiability* is a sufficiently strong guarantee in practice, *i.e.*, the correctness of any returned results is (cryptographically) verifiable. It allows the client to detect any faults with non-reputable evidence, and allows the cloud service provider to retain a formal proof for its correct operations. There has already been a large body of work in the field of providing verifiability for cloud databases [4, 6, 27, 34, 36, 51]. Notably, a classic approach leverages cryptographic primitives to verify the query result of a specific query [30] or an arbitrary SQL [50, 51], but with either limited capability or poor performance.

The emergence of TEE provides a new avenue towards verifiable database. Such trust hardwares act as an additional trust anchor, allowing great simplification and, in turn, performance improvement [4]. In consequence, we build *VeriDB* [53], an SGX-based verifiable database that supports relational tables, multiple access methods and general SQL queries. In *VeriDB*, the client interacts with a query engine that resides in an SGX enclave. Hence, the returned query result can be trusted and easily verified (by checking whether it is endorsed by the SGX), as long as the inputs to the query engine (*i.e.*, the data retrieved from the storage) are correct. This effectively reduces the problem of verifying the query correctness to that of verifying the integrity of data retrieval from the storage.

8 CONCLUSION

Cloud databases have evolved significantly, yet many more important and interesting features are still to be explored and developed. With the rapid development of cloud infrastructures and AI technologies, deep and seamless integration of cloud databases with both the cloud platform and AI techniques are only to be further intensified. By fully leveraging the benefits offered by the underlying cloud infrastructure and the advanced AI techniques, we may soon expect running cloud databases just like playing with legos.

ACKNOWLEDGMENTS

This paper is a summary of the collective work done by the Alibaba Cloud's ApasaraDB database team, in particular, the PolarDB team. The author is grateful to the entire database team, and in particular, to Wei Cao, Zongzhi Chen, Qingda Hu, Gui Huang, Jian Tan, Jianying Wang, Sheng Wang, Jimmy Xingjun Yang, Guangzhou Zhang, Yingqiang Zhang, Wenchao Zhou, and many fellow team members.

REFERENCES

[1] [n.d.]. Working with Aurora Multi-master Clusters. https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-multi-master.html.

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. 2015. Challenges to Adopting Stronger Consistency at Scale. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[3] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. 2020. Azure SQL database always encrypted. In *Proceedings of the 2020 International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1511–1525. https://doi.org/10.1145/3318464.3386141

[4] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 251–266.

[5] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. Speicher: Securing LSM-Based Key-Value Stores Using Shielded Execution. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST '19)*. USENIX Association, USA, 173–190.

[6] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL Engine with Practical Query Authentication. *Proc. VLDB Endow.* 6, 7 (2013), 529–540.

[7] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. 2011. Apache Hadoop Goes Realtime at Facebook. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. Association for Computing Machinery, New York, NY, USA, 1071–1080.

[8] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.

[9] Shaosheng Cao, XinXing Yang, Cen Chen, Jun Zhou, Xiaolong Li, and Yuan Qi. 2019. TitAnt: Online Real-Time Transaction Fraud Detection in Ant Financial. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2082–2093.

[10] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. 2020. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 29–41.

[11] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An Ultra-Low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database. *Proc. VLDB Endow.* 11, 12 (Aug. 2018), 1849–1862. https://doi.org/10.14778/3229863.3229872

[12] Wei Cao, Zhenjun Liu, Peng Wang, Sen Chen, Caifeng Zhu, Song Zheng, Yuhui Wang, and Guoqing Ma. 2018. PolarFS: An ultralow latency and failure resilient distributed file system for shared storage cloud database. *Proceedings of the VLDB Endowment* 11, 12 (2018), 1849–1862.

[13] 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 *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. 2477–2489.

[14] Nathan Chenette, Kevin Lewi, Stephen A Weis, and David J Wu. 2016. Practical order-revealing encryption with limited leakage. In *International conference on fast software encryption*. Springer, 474–493.

[15] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[16] Microsoft Corporation. 2022. Row-Level Security. Retrieved March 1, 2022 from https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security

[17] Oracle Corporation. 2022. Virtual Private Database. Retrieved March 1, 2022 from https://www.oracle.com/database/technologies/virtual-private-db.html

[18] Victor Costan and Srinivas Devadas. 2016. Intel sgx explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[19] Lei Deng, Jerry Gao, and Chandrasekar Vuppalapati. 2015. Building a Big Data Analytics Service Framework for Mobile Advertising and Marketing. In *First IEEE International Conference on Big Data Computing Service and Applications, BigDataService 2015, Redwood City, CA, USA, March 30 - April 2, 2015*. IEEE Computer Society, 256–266.

[20] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1534–1547.

[21] Craig Gentry. 2009. Fully Homomorphic Encryption Using Ideal Lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing* (Bethesda, MD, USA) *(STOC '09)*. Association for Computing Machinery, New York, NY, USA, 169–178. https://doi.org/10.1145/1536414.1536440

[22] Xiao He, Ye Li, Jian Tan, Bin Wu, and Feifei Li. 2023. OneShotSTL: One-Shot Seasonal-Trend Decomposition For Online Time Series Anomaly Detection And Forecasting. *Proc. VLDB Endow.* 16, 6 (apr 2023), 1399–1412.

[23] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-Scale E-Commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 651–665. https://doi.org/10.1145/3299869.3314041

[24] Advanced Micro Devices Incorporated. 2005. Secure Virtual Machine Architecture Reference Manual.

[25] Rodrigo Laigner, Yongluan Zhou, Marcos Antonio Vaz Salles, Yijian Liu, and Marcos Kalinowski. 2021. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *arXiv preprint arXiv:2103.00170* (2021).

[26] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2263–2272. https://doi.org/10.14778/3352063.3352141

[27] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. ACM, 121–132.

[28] Ji You Li, Jiachi Zhang, Wenchao Zhou, Yuhang Liu, Shuai Zhang, Zhuoming Xue, Ding Xu, Hua Fan, Fangyuan Zhou, and Feifei Li. 2023. Eigen: End-to-end Resource Optimization for Large-Scale Databases on the Cloud. In *International Conference on Very Large Data Bases (VLDB)*.

[29] Ye Li, Jian Tan, Bin Wu, Xiao He, and Feifei Li. 2024. ShapleyIQ: Influence Quantification by Shapley Values for Performance Debugging of Microservices. (2024).

[30] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings (Lecture Notes in Computer Science)*, Vol. 293. Springer, 369–378.

[31] Gilad Mishne, Jeff Dalton, Zhenghua Li, Aneesh Sharma, and Jimmy Lin. 2013. Fast data in the era of big data: Twitter's real-time related query suggestion architecture. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1147–1158.

[32] Oracle. [n.d.]. InnoDB. https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html.

[33] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques* (Prague, Czech Republic) *(EUROCRYPT '99)*. Springer-Verlag, Berlin, Heidelberg, 223–238.

[34] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan. 2005. Verifying Completeness of Relational Query Results in Data Publishing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 407–418.

[35] Vasilis Pappas, Fernando Krell, Binh Vo, Vladimir Kolesnikov, Tal Malkin, Seung Geol Choi, Wesley George, Angelos Keromytis, and Steve Bellovin. 2014. Blind Seer: A Scalable Private DBMS. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (S&P '14)*. IEEE Computer Society, USA, 359–374. https://doi.org/10.1109/SP.2014.30

[36] Yanqing Peng, Min Du, Feifei Li, Raymond Cheng, and Dawn Song. 2020. FalconDB: Blockchain-based Collaborative Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*. ACM, 637–652.

[37] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 85–100. https://doi.org/10.1145/2043556.2043566

[38] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy (S&P '18)*. IEEE Computer Society, USA, 264–278.

[39] Kui Ren, Yu Guo, Jiaqi Li, Xiaohua Jia, Cong Wang, Yajin Zhou, Sheng Wang, Ning Cao, and Feifei Li. 2020. HybrIDX: New Hybrid Index for Volume-hiding Range Queries in Data Outsourcing Services. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 23–33. https://doi.org/10.1109/ICDCS47774.2020.00014

[40] Xuanle Ren, Le Su, Zhen Gu, Sheng Wang, Feifei Li, Yuan Xie, Song Bian, Chao Li, and Fan Zhang. 2023. HEDA: Multi-Attribute Unbounded Aggregation over Homomorphically Encrypted Database. *Proc. VLDB Endow.* 16, 4 (dec 2023), 601–614. https://doi.org/10.14778/3574245.3574248

[41] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proceedings of the VLDB Endowment* 14, 6 (Feb. 2021), 1019–1032. https://doi.org/10.14778/3447689.3447705

[42] MELLANOX TECHNOLOGIES. [n.d.]. RDMA aware networks programming user manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[43] The Transaction Processing Council. 2007. TPC-E Benchmark. http://tpc.org/tpce/. "[accessed-April-2022]".

[44] 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 and Informatics* 33, 3 (2016), 793–807.

[45] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, Chengjun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2 (2023), 199:1–199:25. https://doi.org/10.1145/3589785

[46] Sheng Wang, Yiran Li, Huorong Li, Feifei Li, Chengjin Tian, Le Su, Yanshan Zhang, Yubing Ma, Lie Yan, Yuanyuan Sun, et al. 2022. Operon: An encrypted database for ownership-preserving data management. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3332–3345.

[47] 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 (aug 2020), 3152–3165.

[48] Wikipedia. 2015. Consistency Model. https://en.wikipedia.org/wiki/Consistency_model.

[49] Xinjun Yang, Yingqiang Zhang, Hao Chen, Chuan Sun, Feifei Li, and Zhou Wenchao. 2023. PolarDB-SCC: A Cloud-Native Database Ensuring Low Latency for Strongly Consistent Reads. *Proc. VLDB Endow.* 16, 12 (Aug. 2023).

[50] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL Queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 863–880.

[51] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for Outsourced Databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*. ACM, 1480–1491.

[52] Jun Zhou, Xiaolong Li, Peilin Zhao, Chaochao Chen, Longfei Li, Xinxing Yang, Qing Cui, Jin Yu, Xu Chen, Yi Ding, and Yuan Alan Qi. 2017. KunPeng: Parameter Server Based Distributed Learning Systems and Its Applications in Alibaba and Ant Financial. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. Association for Computing Machinery, New York, NY, USA, 1693–1702.

[53] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-Based Verifiable Database *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2182–2194. https://doi.org/10.1145/3448016.3457308