PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba

JIANYING WANG, TONGLIANG LI, HAOZE SONG, XINJUN YANG, WENCHAO ZHOU, FEIFEI LI, BAOYUE YAN, QIANQIAN WU, and YUKUN LIANG, Alibaba Group, China
CHENGJUN YING, Alibaba Group, China and Zhejiang University, China
YUJIE WANG, BAOKAI CHEN, CHANG CAI, YUBIN RUAN, XIAOYI WENG, SHIBIN CHEN, LIANG YIN, CHENGZHONG YANG, XIN CAI, HONGYAN XING, NANLONG YU, XIAOFEI CHEN, and DAPENG HUANG, Alibaba Group, China
JIANLING SUN, Alibaba Group, China and Zhejiang University, China

Cloud-native databases have become the de-facto choice for mission-critical applications on the cloud due to the need for high availability, resource elasticity, and cost efficiency. Meanwhile, driven by the increasing connectivity between data generation and analysis, users prefer a single database to efficiently process both OLTP and OLAP workloads, which enhances data freshness and reduces the complexity of data synchronization and the overall business cost.

In this paper, we summarize five crucial design goals for a cloud-native HTAP database based on our experience and customers' feedback, i.e., transparency, competitive OLAP performance, minimal perturbation on OLTP workloads, high data freshness, and excellent resource elasticity. As our solution to realize these goals, we present PolarDB-IMCI, a cloud-native HTAP database system designed and deployed at Alibaba Cloud. Our evaluation results show that PolarDB-IMCI is able to handle HTAP efficiently on both experimental and production workloads; notably, it speeds up analytical queries up to ×149 on TPC-H (100*GB*). PolarDB-IMCI introduces low visibility delay and little performance perturbation on OLTP workloads (< 5%), and resource elasticity can be achieved by scaling out in tens of seconds.

CCS Concepts: • **Information systems → DBMS engine architectures**; *Database transaction processing*; *Online analytical processing engines.*

Additional Key Words and Phrases: cloud databases, hybrid transactional and analytical processing

ACM Reference Format:
Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, Chengjun Ying, Yujie Wang, Chang Cai, Baokai Chen, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, Jianling

Authors' addresses: Jianying Wang, beilou.wjy@alibaba-inc.com; Tongliang Li, litongliang.ltl@alibaba-inc.com; Haoze Song, songhaoze.shz@alibaba-inc.com; Xinjun Yang, xinjun.y@alibaba-inc.com; Wenchao Zhou, zwc231487@alibaba-inc.com; Feifei Li, lifeifei@alibaba-inc.com; Baoyue Yan, baoyue.yby@alibaba-inc.com; Qianqian Wu, daisy.wqq@alibaba-inc.com; Yukun Liang, liangyukun.lyk@alibaba-inc.com, Alibaba Group, China; Chengjun Ying, yingcj@zju.edu.cn, Alibaba Group, China and Zhejiang University, China; Yujie Wang, zhencheng.wyj@alibaba-inc.com; Baokai Chen, baokai.cbk@alibabainc.com; Chang Cai, caichang.cc@alibaba-inc.com; Yubin Ruan, yubin.ryb@alibaba-inc.com; Xiaoyi Weng, echo.wxy@alibaba-inc.com; Shibin Chen, wuha.csb@alibaba-inc.com; Liang Yin, allen.yinl@alibaba-inc.com; Chengzhong Yang, chengzhong.ycz@alibaba-inc.com; Xin Cai, frank.cx@alibaba-inc.com; Hongyan Xing, diane.xhy@alibaba-inc.com; Nanlong Yu, nanlong.ynl@alibaba-inc.com; Xiaofei Chen, chenxiaofei.cxf@alibaba-inc.com; Dapeng Huang, wuzang.hdp@alibabainc.com, Alibaba Group, China; Jianling Sun, sunjl@zju.edu.cn, Alibaba Group, China and Zhejiang University, China.

2836-6573/2023/6-ART199
https://doi.org/10.1145/3589785

Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data* 1, 2, Article 199 (June 2023), 25 pages. https://doi.org/10.1145/3589785

1 INTRODUCTION

In recent years, cloud-native databases [8, 21, 26, 53] have become an inexorable trend in the database industry. Different from on-premise databases, a cloud-native database decouples its architecture into two layers: a computation layer and a storage layer, allowing resources to scale independently. Nodes equipped with disks (in the storage layer) form a shared storage pool that serves as a unified data interface for nodes in the computation layer. This disaggregation architecture enables database systems to offer extreme elasticity, flexible on-demand charging models, and low operating costs for customers. As a result, the market of cloud-native databases has quickly taken off [37].

Meanwhile, we have witnessed another trend that the line between classic OLTP and OLAP databases started to blur: there is a growing need for a database to provide sufficient support for *both* transactional processing and analytical processing, especially in the fields of business intelligence [52], social media [5, 39], fraud detection [7], and marketing [22, 57]. To provide such capability, traditional solutions often deploy data and application logic into two databases, one specialized in OLTP and the other in OLAP (e.g., MySQL [41] for OLTP, and ClickHouse [15] for OLAP), and rely on data synchronization techniques (such as Extract-Transform-Load [51] (ETL) workflow) for ensuring consistencies between them, as shown in Figure 1. According to our statistics, nearly 30% of the customers of PolarDB, an OLTP database, synchronize data to an independent data warehouse system for data analytics needs.

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. In practice, these issues lead to sub-optimal user experience and a large number of user inquiries. To address these issues, it calls for a cloud-native Hybrid Transactional and Analytical Processing (HTAP) database. In this paper, we present PolarDB-IMCI, a cloud-native HTAP database deployed at Alibaba Cloud. We summarize the crucial design goals of PolarDB-IMCI below, which are also applicable to the design of a general cloud-native HTAP database.

- **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 two *isolated* systems (e.g., engines, indexes, interfaces, etc.) for OLAP and OLTP queries respectively. Our system should provide a unified SQL interface for both OLAP and OLTP workloads.
- **G#2: Advanced OLAP Performance.** As a major goal of any HTAP database, the OLAP performance (e.g., execution latency) of PolarDB-IMCI should be comparable to typical databases specialized in processing OLAP queries (typically through the introduction of columnar data storage).
- **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. In this paper, we follow earlier similar work [12, 28] using the visibility delay as a

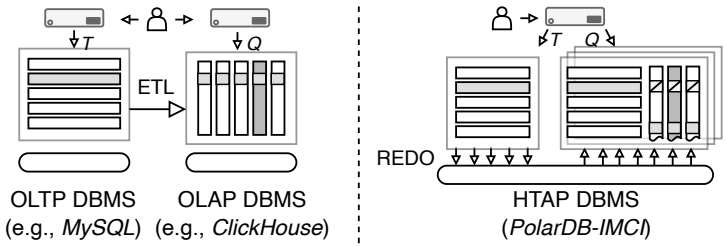


Fig. 1. Comparison of ETL and PolarDB-IMCI.

freshness score for a query. By definition, the visibility delay is the time interval during which updates to the database can be visible to OLAP queries.

- **G#5: Excellent Resource Elasticity.** In HTAP scenarios, the consumption of CPU/IO resources fluctuates significantly, from hundreds to thousands of times. As a key feature of cloud-native databases, our system should ensure high resource elasticity (e.g., scale-out in minutes or even seconds) to adaptively serve the changing data volume and analytical workloads with stable performance and high resource utilization.

PolarDB-IMCI meets all desired goals (i.e., **G#1-5**) with the following innovations. First, to meet G#1 and G#2, we implemented *in-memory column index (IMCI, §4)* as complementary storage. PolarDB-IMCI absorbs diverse advanced optimizations from the OLAP community and derives *a new SQL engine (§6.3)* to match the execution mode on columns. Further, PolarDB-IMCI proposes *a new query routing mechanism (§6.1)* that dispatches queries transparently.

Second, to meet G#3, PolarDB-IMCI resides column indexes on *separated read-only (RO) nodes (§3.1)* with a shared storage architecture to provide effective resource isolation between OLTP and OLAP requests. Updates are propagated to RO nodes by *reusing REDO logs (§5.3)* (i.e., the differential logging for the row store) instead of shipping additional logical logs (i.e., MySQL Binlogs).

Third, to meet G#4, we enhance our update propagation framework with *commit-ahead log shipping (CALS, §5.1)* and *2-Phase conflict-free log replay (2P-COFFER, §5.2)*. CALS ships transaction logs before committing. 2P-COFFER efficiently parses and applies REDO logs to RO nodes. Furthermore, we implemented the column index as *append-only storage (§4)*: records are organized in insert order rather than primary key order. Thus, updates to column indexes are performed out-place and quickly.

Finally, to meet G#5, the checkpoint mechanism of the columnar index is seamlessly built into PolarDB's original storage engine. Therefore fast scale-out capability can be achieved by quickly pulling up a RO node using the checkpoint on shared storage (§7).

We started the design and development of cloud-native PolarDB in 2017, and seek for an HTAP solution (i.e., PolarDB-IMCI) in 2019. By now, PolarDB-IMCI is severing a large number of internal and external customers (Table 3). The key contributions of this work are listed as follows:

- We propose PolarDB-IMCI, an HTAP solution for cloud-native relational database systems. To the best of our knowledge, PolarDB-IMCI is the first cloud-native HTAP database to satisfy all of the aforementioned design goals.
- We design an architecture that provides dual-format storage on read-only nodes under the storage-computation separation architecture, which enables efficient execution of analytical queries and minimizes the impact on OLTP load. Additionally, PolarDB-IMCI is the first practical template to demonstrate that it is possible and applicable to implement replication from row-store to dual-format storage with physical redo logs while reducing replication latency to millisecond levels.

- We evaluate PolarDB-IMCI with diverse experiments (in both experimental and production environments). The experimental results show that PolarDB-IMCI outperforms row-based PolarDB up to ×149 on a standard analytical workload TPC-H (100*GB*), and its performance is comparable to the advanced OLAP databases (e.g., ClickHouse). Performance degradation on OLTP is tiny (less than 5%), even when OLAP workloads increase continuously. The visibility delay of PolarDB-IMCI at is <5*ms* on typical workloads, and <30*ms* under heavy workloads. PolarDB-IMCI can scale out in tens of seconds.

The remainder of the paper is organized as follows. §2 introduces the background of HTAP and cloud-native databases. §3 presents the architecture. PolarDB-IMCI's components and update propagation framework are introduced in §4 and §5 respectively. §6 discusses query dispatch, optimization, and execution. §7 introduces the checkpoint mechanism. §8 details the experiments and evaluation. §9 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 Hybrid Transactional/Analytical Processing

For long decades, OLTP and OLAP databases are dedicatedly designed for their respective workloads. For instance, OLTP engines (e.g., *MySQL* [41]) prefer row-based data formats, row-at-a-time operators, and early materialization strategy, favoring data modification and point queries. On the contrary, OLAP engines (e.g., ClickHouse [15]) use column-based data formats, batch-at-a-time operators, and late materialization strategy, favoring scan-intensive analytical queries. As a result, modern database administrators often need to deploy both OLTP and OLAP databases, and conduct data shipping between two types of databases (e.g., ETL [51]).

The emergence of HTAP databases eliminates the burden of maintaining multiple databases and simplifies data shipping. We classify existing HTAP solutions into two categories (i.e., single-instance and multi-instance), and discuss each category below.

HTAP with Single Instance. SAP HANA [48] supports hybrid workloads by introducing a three-tier merge tree, a layered in-memory store that supports both row and column formats. Oracle Dual [31] allows relational tables to be built as In-Memory Column Units (IMCU) to provide fast column scans. New updates to IMCUs are temporarily logged by metadata, and IMCUs can be repopulated from the memory buffer when more updates are accumulated. Unlike Oracle Dual, SQL Server CSI [33, 34] supports column stores with column store index (CSI) and periodically merges new updates into CSI, thus eliminating rebuilding.

PolarDB-IMCI follows a similar principle, but pioneers this design to the cloud-native architecture by addressing a number of key challenges as detailed in later sections.

HTAP with Multiple Instance. Another type of HTAP database utilizes replication techniques to maintain multiple instances. Thus, transactional and analytical queries can be routed to different instances to achieve efficient performance isolation. Further, each instance can tailor its architecture to fit workloads.

A more recent work of SAP HANA proposes Asynchronous Table Replication (ATR) [35] for data synchronization between the primary instance and replicas. Replication logs are supplied asynchronously to replicas and are replayed in parallel in session granularity. Unlike ATR, Google F1 Lightning [55] uses Change Data Capture (CDC), a more loosely coupled mechanism shuffling data via BigTable. TiDB [28] uses Raft [42] to connect row-store engines (TiKV) and columnar engines (TiFlash). TiFlash behaves as a Raft learner receiving logs asynchronously from the leader and does not participate in the leader election. IBM DB2 Analytics Accelerator (IDAA) [6] maintains a copy of row-based table data by integrated synchronization to support incremental updates. A new version of Oracle Dual [44] supports offloading read-only workloads to homogeneous instances

(standby) and synchronizes data by REDO logs. ByteHTAP [12] uses disaggregated storage and synchronizes heterogeneous engines (ByteNDB for OLTP and Apache Flink [18] for OLAP) by Binlog. Different from these works, PolarDB-IMCI directly reuses REDO logs for heterogeneous data replication. To the best of our knowledge, PolarDB-IMCI is the first industrial database using physical logs to efficiently synchronize heterogeneous storage.

Additionally, several databases leverage shared storage for data synchronization. Wildfire [2] is a Spark-compatible database. It uses SparkSQL as the engine for analytical processing. Data updates are first committed to the local SSD and then moved asynchronously to its shared storage. PolarDB-IMCI also adopts a shared storage (i.e., PolarFS [8]), but supports real-time synchronization.

2.2 Cloud-Native Database

The key technique of cloud-native databases is decoupling computation and storage. A typical cloud-native database often adopts cloud storage underneath its storage engine, leveraging another layer for virtualization and providing an elastic storage service [14]. Cloud-native architecture benefits customers with high resource elasticity and an on-demand charging model and benefits service providers by reducing maintenance and development costs.

Cloud-native OLTP/OLAP. Aurora [53, 54] is a cloud-native OLTP database deployed on a custom-designed cloud storage layer. Taurus [23] also separates the compute and storage layers in a similar manner but uses asymmetric replication based on separate persistence mechanisms for database logs and pages.

Besides OLTP systems, OLAP databases also benefit from storage-disaggregation. Several conventional data warehousing systems have adapted to the cloud (e.g., Vertica [32], Eon [50]), and several OLAP databases are natively developed for the cloud (e.g., Snowfake [21], Redshift [26], AnalyticDB [56]).

Cloud-native HTAP. SingleStore [45] takes the first step to make the HTAP database cloud-native. It disaggregates computation and storage, and supports committing transactions on the local disk of computation nodes and pushing data asynchronously to its blob storage. Different from SingleStore, PolarDB-IMCI offloads all persisted data into the shared storage layer, thus all states of the computation nodes can be rebuilt from shared storage directly, favoring recovery and elasticity.

3 OVERVIEW

In this section, we first outline the architecture of PolarDB-IMCI, then summarize the design rationales driven by the aforementioned design goals, along with a brief description of the user interface.

3.1 Architecture of PolarDB-IMCI

Figure 2 shows the architecture of PolarDB-IMCI, which follows the crucial design principle of separating computation and storage architecture. The storage layer is a user-space distributed file system called PolarFS [8] with high availability and reliability. The computation layer contains multiple computation nodes, including a primary node for read/write requests (RW node), several nodes for read-only requests (RO nodes), and several stateless proxy nodes for load balancing. Given this, PolarDB-IMCI can provide high resource elasticity (§7). Furthermore, all nodes in both storage and computation layers are connected by a high-speed RDMA network to achieve low latency of data access.

To speed up analytical queries, PolarDB-IMCI supports building in-memory column indexes (§4) 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

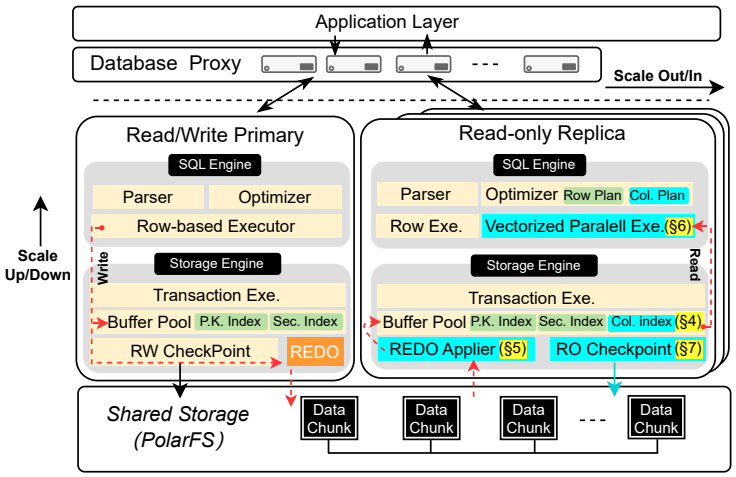


Fig. 2. Cloud-native architecture of PolarDB-IMCI.

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 (§5) 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 other strawmen approaches (e.g., using Binlog). 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 execution engines (§6): 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. PolarDB-IMCI's optimizer automatically generates and coordinates plans for both execution engines, which is transparent to the consumer.

3.2 Design Rationales

We highlight the design rationales of PolarDB-IMCI below, which may also apply to other cloud-native HTAP databases.

Storage-Computation Separation. As a key design principle of cloud-native databases, the storage-computation separation architecture enables adaptive compute resource provisioning to shifting workloads without data movement, which has become a mainstream architecture alternative. PolarDB-IMCI takes the decision to naturally match our design goal **G#5** (high resource elasticity).

Single RW Nodes with Multiple RO Nodes. As a practical design decision, single-writer architecture has been confirmed to have advanced write performance [53] and significantly reduce the system complexity. We have observed that a single RW node is enough to serve 95% customers in

```
CREATE TABLE demo_table {
        C1 INT(11) NOTNULL,
        C2 INT(11) DEFULT NULL,
        C3 INT(11) DEFULT NULL,
        C4 INT(11) DEFULT NULL,
        C5 LONGTEXT DEFULT NULL,
        PRIMARY KEY(C1),
        KEY SEC_INDEX(C2),
        KEY COLUMN_INDEX(C3, C4, C5)
        }
```

Fig. 3. A DDL creates a demo table with a primary key index on C1, a secondary index on C2, and column indexes on column C3,C4,C5.

our business. With the design choice, all RO nodes have a consistent data view synchronized with the RW node. Large OLAP queries are routed to RO nodes to enable efficient resource isolation and the RO nodes can be quickly scaled out to serve surging OLAP queries, which follows the design goal **G#3** (minimal perturbation on OLTP) and **G#5** (resource elasticity).

Hybrid Execution and Storage Engines inside RO Nodes. From the lessons in the OLAP community, columnar data layout and vectorized batch execution are significant optimizations for OLAP queries. However, it is not a wise decision for us to use an existing column-oriented system (e.g., ClickHouse) to serve directly as RO nodes. There are two reasons for this. First, it is time-consuming to achieve full compatibility between the RW node and RO nodes. In a cloud service environment, even little incompatibility can be drastically amplified and overwhelm developers given the huge customer volume. Second, pure column-oriented RO nodes are still inefficient for point-lookup queries, which are classified as OLTP workloads. As a result, we started to design a new column-based execution engine extending the original execution engine of PolarDB, to satisfy the goal **G#1** (transparency). The column-based execution engine is designed to meet **G#2** (advanced OLAP performance). While the row-based execution engine handles incompatible and point-lookup queries that the former cannot deal with. RO nodes have both column-based and row-based execution and storage engines.

Dual-format RO Nodes Synchronized by Physical REDO Logs. With the architecture over the shared storage, new RO nodes can be quickly started to serve surging read-only queries to meet the design goal **G#5**, and can continuously apply REDO logs from the RW node to keep storage fresh (i.e., **G#4**). However, synchronizing heterogeneous storage with the original physical logs (i.e., REDO logs) is challenging as the logs are tightly coupled with the underlying data structures (e.g., pages). Therefore, a strawman approach is letting the RW node record additional logical logs (e.g., Binlog) for column-store. The drawback is significant: it triggers additional *fsyncs* when committing a transaction, thus causing non-negligible performance perturbation on OLTP. Given this, we dedicatedly designed a new synchronization method by reusing REDO and making up logical operations from physical logs on RO nodes. It is feasible since PolarDB-IMCI maintains both row-based buffer pool and column indexes on RO nodes. Logical operations can be regained by the applying process on a row-based buffer pool. Our evaluation shows that the overhead of reusing REDO logs is significantly lower than using Binlog.

3.3 User Interface

Column store in PolarDB-IMCI is exposed as a new index type: column index. Applications can create a column index for a table on demand. As PolarDB-IMCI is fully compatible with *MySQL*,

applications can use the SQL statement with *MySQL* syntax to create a column index. An example is shown in Figure 3. It creates a table with five columns, the primary key index is created on column C1, a secondary index is created on column C2, and column indexes are created on columns C3, C4, and C5.

In addition, to specify the columns included in column indexes when creating the table, applications may also use the ALTER statement to add a column index later. When applications execute Data Definition Language (DDL) on a table with a large number of rows to add a column index. The RO node will issue a consistent read on PolarDB-IMCI's row store, scan the checkpoint, and convert it to a column index in parallel. Note that adding column indexes in PolarDB-IMCI is an online operation: the queries and DML operations on the table can process together while a DDL operation is in progress. The changes made by concurrent DML operations will be recorded in a buffer and applied to the new column index at the end of the process.

4 COLUMN INDEX STORAGE

This section dives into the column index store, a crucial part of PolarDB-IMCI for handling analytical queries. PolarDB-IMCI supports row-based storage engines [14, 29] that are highly tuned for transaction processing on cloud storage. However, row-based data formats are well known for being inefficient to serve analytical queries. Inspired by pioneering industrial databases (e.g., Oracle [31], SQL Server [33]), PolarDB-IMCI implements a dual data format via in-memory column indexes, to enhance OLAP functionality.

4.1 Data Organization of Column Index

As shown in Figure 4, column indexes in PolarDB-IMCI serve as complementary storage to the existing row store. In PolarDB-IMCI, columns of a table can selectively be involved in a column index. PolarDB-IMCI divides all rows of a table into multiple row groups with append-only writes to improve the write performance. In a row group, each column of data is organized into a data Pack, along with some metadata for statistics. To provide snapshot isolation, each row group contains an insert Version Id (VID) map, and a delete VID map to control the visibility for concurrent transaction processing. Since the row groups are append-only, deletes require an explicit row id for the given primary key to set the delete version for that row. To realize it, PolarDB-IMCI implements a Row-ID locator (i.e., a two-layered LSM tree) to map the primary key to the physical position of the row in the column index.

Data Pack Layout. A relational table is first divided into multiple row groups with configurable size (i.e., 64K rows per row groups), and the left rows form a partial row group (e.g., Row Group N in Figure 4). To realize fast data ingestion, row groups are append-only (§4.2). That is, the full-sized row groups are immutable, and partial row groups will be fulfilled in an append-only manner. The data belonging to the same column within a row group is organized as a Data Pack in a compressed format to reduce space consumption. Note that PolarDB-IMCI does not compress Partial Packs as they are updated continuously.

Pack Meta. To avoid unnecessary data access during query execution, PolarDB-IMCI maintains a Pack meta for each Data Pack. The Pack meta keeps track of minimum and maximum values as well as a sampling histogram for each Pack, which benefits column scan. For instance, when a query statement specifies a WHERE clause predicate, Pack meta for the referenced column can be used to check whether the scan on this Pack can be skipped.

RID Locator. As the data in Packs is stored in its insertion order, PolarDB-IMCI relies on a locator to map primary keys to their corresponding physical locations in column indexes. In PolarDB-IMCI, each row is assigned an increasing and unique Row-ID (RID) by its insertion order. Then, the RID locator records the mapping of Key-Values pairs (i.e., <Primary Key, RID>). Delete operations rely

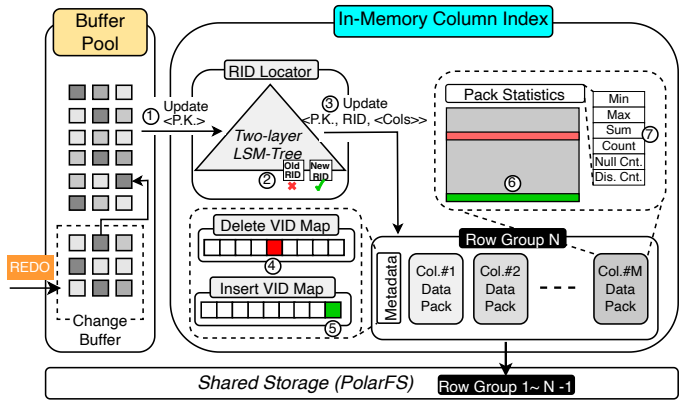


Fig. 4. This diagram shows how data is updated in IMCI storage (i.e., step ①~⑦). For simplicity, both delete and insert operations are performed in the last column data Pack (i.e., partial Packs). "RID" means row id. "VID" means version id.

on the locator to find the physical position of records. PolarDB-IMCI uses a two-layered LSM tree for the RID locator. Compared to other data structures, the LSM tree helps the locator achieve near-optimal memory utilization and easily extends to disks.

Version Id (VID) Map. PolarDB-IMCI uses Multi-Version Concurrency Control (MVCC) to provide consistent data views. For column indexes, updating a record is appending a new version of this record to the tail of Partial Packs. The old version of the record is logically deleted by marking it with a timestamp. Each version has a 64-bit insert VID and a delete VID, recording the timestamps of the appending and deleting of this version, respectively. A read transaction determines a version is visible by checking its timestamp is within the range of the insert VID and the delete VID. All insert (delete) VIDs in a row group form an insert (delete) VID map.

4.2 DML Operation on Data Packs

To better understand the process flow on data Packs, we now describe how to conduct DML operations on the data structure of column indexes.

- **Insert:** Inserting a row into a column index consists of the following four steps. First, the column index allocates an empty RID from its Partial Packs. Second, the locator updates the new RID by the primary key for the inserted row (i.e., add a new record into the LSM tree). Then, the column index writes row data into the empty slot (e.g., data Packs within the Row Group N in Figure 4). Finally, the insert VID records the transaction committed sequence number (i.e., timestamp) of the inserted data. Since the insert VID map maintains the insert version of each inserted data, it also follows the append-only write pattern.
- **Delete:** The delete operation retrieves a row's RID via the RID locator by its primary key (PK) and then sets the corresponding delete VID with its transaction committed sequence number. After that, the mapping between the PK and RID is removed from the locator to ensure data consistency.
- **Update:** As shown in Figure 4, an update on the column index is performed as a delete operation followed by an insert operation. The updated version of a row is appended to Partial Packs, and the old version is logically deleted from its original data Pack (i.e., set the delete VID to max value).

As a result, column indexes are arranged in insertion order with fast data ingestion. Another significant benefit of out-place updates is that it avoids the contention for modification of the same row (§5.4).

4.3 Data Pack Compression and Compaction

Compression. A Partial Pack is transformed into a Pack when it reaches its maximum capacity and then compressed into disks to reduce space consumption. The compression process is carried out with a copy-on-write pattern to avoid access contentions. That is, a new Pack is generated to hold the compressed data, with no changes to the Partial Pack. PolarDB-IMCI updates the metadata after compression to replace the Partial Pack with the new Pack (i.e., atomically updating the pointer to the new Pack). For the various data types, column indexes employ different compression algorithms. Numerical columns adopt the combination of frame-of-reference, delta-encoding, and bit-packing compression, and string columns use dictionary compression.

Additionally, since Packs are immutable, the insert VID map of that Pack is useless when active transactions are greater than all VIDs, i.e., no active transactions refer to the insert VID map. In such cases, PolarDB-IMCI removes the insert VID maps in row groups to reduce memory footprint.
Compaction. Delete operations may set delete VIDs in a Pack, which punches holes for that Pack. As the number of invalid rows increases over time, the scan performance and the space efficiency degrade. PolarDB-IMCI periodically detects and re-arranges under-flowing Packs to keep a low waterline for invalid rows of column indexes. For example, sparse Packs, with less than half of the valid rows, are picked as under-flowing. Then the background threads issue a compaction transaction, which includes numerous update operations, one for each migrated valid row, to re-append all valid rows of picked Packs into Partial Packs. Recall that the update operations of column indexes are out-place, so the old rows are still accessible for foreground operations during or even after the compactions, which enables non-blocking updates. The picked Packs after compactions will be permanently removed when no active transaction accesses them.

5 UPDATE PROPAGATION

In this section, we describe our efforts for synchronizing heterogeneous data storage. Minimal perturbation on OLTP is a high-priority goal for PolarDB-IMCI. To achieve this goal, update propagation in PolarDB-IMCI is implemented by REDO logs, eliminating the overhead for RW to persist additional logical logs. On top of REDO logging, PolarDB needs to keep RO nodes as up-to-date as possible for data freshness. For this purpose, we introduce Commit-Ahead Log Shipping (CALS) to reduce visible delay and 2-Phase COnFlict-Free parallEl Replay (2P-COFFER) mechanism to improve replay throughput.

5.1 Commit-Ahead Log Shipping

To minimize performance perturbation, in PolarDB-IMCI, updates to RO nodes are fully asynchronous without affecting RW transaction commits. Given this, to enhance data freshness, PolarDB-IMCI uses the CALS technique, which ships logs before transaction committing. As illustrated in Figure 5, a transaction consists of multiple log entries:

the last entry is a commit or an abort log, whereas the ones before it is DML logs. Each log entry is assigned a log sequence number (LSN). For example, the transaction with TID 101 has three log entries with LSN 300 $\sim$ 302. Log entries 300 and 301 are DMLs. Log entry 302 contains the decisions on the transaction (i.e., abort).

After the RW node writes a log entry to the shared storage (i.e., PolarFS), it notifies RO nodes by broadcasting its up-to-date LSN (299 in our example). When receiving LSNs, RO reads logs

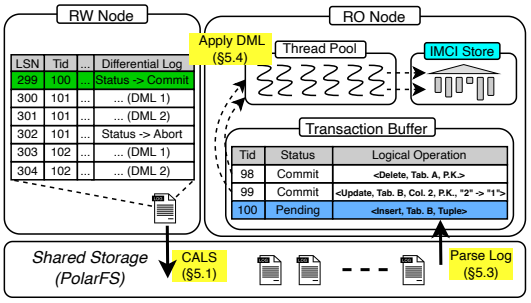


Fig. 5. An overview of REDO Log shipping. Logs are shipped from RW node to RO node by shared storage.

from PolarFS immediately. Each DML log is then parsed into a DML statement and stored in a transaction buffer based on its TID (one buffer unit per transaction).

The whole process does not require waiting for the RW node to commit the transaction. For example, the DMLs in the transaction with TID 100 will ship before the log entry 299 (final commit). When the RO node reads a commit log entry, the earlier DML statements are already parsed and delivered as logical operations in the transaction buffer, allowing PolarDB-IMCI to replay the DMLs immediately. When reading an abort log entry, RO simply frees the transaction buffer and no data need to be rolled back.

5.2 Two-Phase Conflict-Free Parallel Replay

As mentioned previously, PolarDB-IMCI does not generate additional logical logs for update propagation but reuses REDO logs. The reason is that log delivery makes the RW node write more log entries, which affects OLTP performance. However, for a significant period, it is regarded as almost impossible to synchronize heterogeneous storage with REDO logs [35]. There are three challenges to this. (1) REDO logs only record changes to physical pages in the row store and lack database-level or table-level information [43] (e.g., RO nodes do not know which table the page change corresponds to). (2) Page changes caused by the row store itself rather than user DMLs are also included in REDO logs, such as B+tree splits/merges and page consolidations. Column indexes cannot apply these logs, otherwise, inconsistencies may occur. (3) REDO logs only include differences rather than complete updates to reduce log volume.

As shown in Figure 6, PolarDB-IMCI addresses these challenges with two replay phases. The **Phase#1** is to replay REDO logs to an in-memory copy of the row store in RO. In this phase, PolarDB-IMCI captures the complete information to parse REDO logs into logical DML statements. Then, the **Phase#2** is to replay DML statements to column indexes.

The performance of replay is critical to our system. To achieve high performance, several parallel replay mechanisms [6, 46, 47, 55] are proposed in the literature. These works either take parallel replay at session granularity or transaction granularity with the help of conflict-handling aids, such as locks or dependency graphs, or optimistic control. Unlike these works, PolarDB-IMCI proposes a new replay approach, 2P-COFFER, to make both phases of parallel replay conflict-free. In 2P-COFFER, the **Phase#1** is page-grained, while the **Phase#2** is row-grained to enable the concurrent modification of different pages/rows. With 2P-COFFER, the replay throughput of RO nodes is much higher than the OLTP throughput of RW (§8.4).

5.3 Phase#1: Physical Log Parse

As shown in Figure 7, a REDO log entry of PolarDB contains multiple fields. For simplicity, we take the update operation as an example, and other sorts of operations are similar.

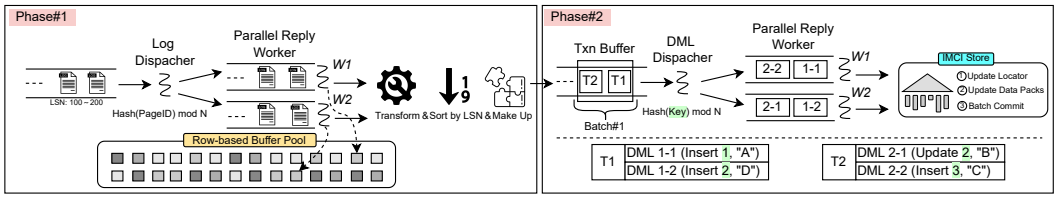


Fig. 6. This diagram shows PolarDB-IMCI's Two-Phase Conflict-Free Parallel Replay (2P-COFFER). In the first phase, REDO logs are parallelly replayed to the row-based buffer pool, parsed to logical DMLs, sorted, and made up to form transactions. In the second phase, DMLs (inside each transaction) are processed in batches. DMLs are dispatched based on their primary keys and update column indexes parallelly.

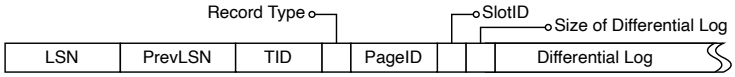


Fig. 7. REDO Log Format.

- TID is the transaction identifier that creates this entry.
- LSN represents the order of this entry in the log.
- PageID identifies which physical page the row updated by this entry belongs to. The Offset field (SlotID) further determines where the updated row sits on the page.
- Data field (Differential Log) contains the difference between the updated value and the original value.

In the left part of Figure 6, **Phase#1** distributes REDO logs to different workers based on the PageID. The distribution process is similar to **Phase#2** (§5.4) but at page granularity. Then, each worker replays its logs in the LSN order to reproduce the DML details. For an update-type log entry, the worker will generate a delete DML and an insert DML during replay since column indexes are updated out-place. The differential field of REDO logs may not contain PK information, which is required for deleting DMLs (find a row via the locator). Therefore, the worker gets the old row from PolarFS based on the PageID and offset field, and uses the old row's PK to assemble a delete-type DML. Then, the worker applies the differential field into the extracted rows to replay page changes, and assemble the insert DML after applying. To truly make up an operation into a logical DML, each operation must also be supplemented with its table schema. Workers get table schema information by table IDs recorded on pages.

Furthermore, workers must identify log entries generated by the row store itself (e.g., B+tree splits). These logs should not be assembled into DMLs. To handle this, workers first check whether a log entry belongs to an active transaction by the TID. If not, this entry is confirmed as not being generated by a user transaction. If so, the worker further checks if the PK of this entry is repeatedly inserted in the active transaction (via a PK set). Note that a duplicate PK insert is not a user DML.

Consequently, reusing REDO forces a replay of all page changes. As an optimization, PolarDB-IMCI let RO nodes maintain the buffer pool of the row store like RW to reduce the amount of data page reads.

In our practice, the computing capacity of **Phase#1** is much greater than the log production capacity of RW. On the one hand, RO nodes directly reproduce page changes without the overhead of redoing transactions, such as B+tree traversals. On the other hand, REDO logs under real workloads always act on hot pages so that the buffer pool has a hit rate close to 99%. Although the buffer pool reduces the memory available for OLAP, we take this tradeoff because reducing the perturbation on OLTP through REDO logs is a higher priority in our scenario.

5.4 Phase#2: Logical DML Apply

REDO logs' LSN order ensures the fundamental prerequisite for log replay, which means changes to RO nodes can be made in the same order as RW. **Phase#1** breaks this order. Therefore, after the parse, a background thread will sort DMLs according to the LSN of their associated log entries. Then, the background thread inserts DMLs into transaction buffer units based on their TID.

In **Phase#2**, a dispatcher distributes a batch of transactions to multiple workers, performing modifications to column indexes in parallel. The distribution is conducted row-by-row, and DML statements from a single transaction will be dispatched to multiple workers for replay. For a DML statement, the dispatcher assigns a specified worker by taking a *modulo* of the hash value of the row's primary key. Therefore, DML statements that modify the same row are assigned to the same worker in the commit order, even if they belong to different transactions. The dispatcher processes each transaction in the commit order, ensuring that different modifications to the same row are delivered to the same worker in order, which guarantees consistency. Each worker follows the steps described in §4.2 to replay each DML statement in order, and changes will be committed to column indexes in batch.

The right part of Figure 6 illustrates how two workers (W_1 and W_2) can replay two transactions (T_1 and T_2) simultaneously. T_1 Insert (1, "A") and Insert (2, "D") respectively. T_2 Update (2, "B") and Insert (3, "C"). Insert (2, "D") and Update (2, "B") are assigned to W_2 with the commit order of T_1 and T_2. W_1 executes these two DMLs in sequence without concurrent conflicts.

5.5 Handle Large Transactions

So far, we have presented the update propagation of PolarDB-IMCI, but there is one more issue. As stated in §5.1, CALS prefetches log entries from PolarFS into transaction buffer units before **Phase#2**. Therefore, if a transaction comprises too many DMLs, its transaction buffer unit may consume a huge memory.

To avoid excessive memory consumption, PolarDB-IMCI pre-commits large transactions: DML statements in a transaction buffer unit are pre-committed when their number reaches a given threshold. The basic idea behind pre-committing is to write updates to Partial Packs with invalid insert and delete VIDs, rendering the updates temporarily invisible. The specific steps of pre-committing are as follows. First, request a continuous RID range for all rows in the current transaction buffer unit, and save this RID range. It is important to note that the global RID locator cannot yet be changed during the pre-commit phase to avoid the exposure of uncommitted transactions. Thus, PolarDB-IMCI creates a temporary RID locator instead of updating the RID global locator to cache new PK-to-RID mappings. Then, PolarDB-IMCI writes the updates to Partial Packs while setting the insert and delete VIDs as invalid to make them invisible. Finally, PolarDB-IMCI frees the memory used by the transaction buffer unit.

When the large transaction commits, PolarDB-IMCI merges the temporary RID locator into the global RID locator and rectifies the invalid VIDs (in the saved RID range) with the transaction commit sequence number. Otherwise, if the large transaction aborts, the temporary locator will be cleaned out. Pre-commit rows remaining in Partial Packs are invalid and will be eliminated later by compaction threads in the background.

6 ANALYTICAL PROCESSING

6.1 Transparent Query Routing

In PolarDB-IMCI, queries can be executed on different nodes and different execution engines via a cost-based routing protocol. The routing process is completely transparent to applications and users and has a two-levels policy: inter-node routing and intra-node routing. Inter-node routing

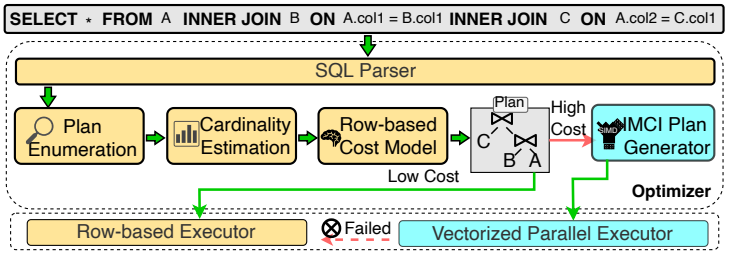


Fig. 8. The workflow of PolarDB-IMCI's optimizer.

implements read/write flow splitting (with load balance) through the proxy layer, while intra-node routing provides a dynamic selection of data access paths and execution engines (either row-based or column-based) through the optimizer.

Inter-node Routing. PolarDB-IMCI's proxy provides a unified SQL interface for all application requests (both OLTP and OLAP). When requests come in, the proxy directs read/write requests (e.g., transactions) to the RW node and directs read-only queries (e.g., analytical queries) to RO nodes via a rough syntax parser. If multiple RO nodes are deployed, the proxy will balance the traffic based on the number of active sessions.

Intra-node Routing. As shown in Figure 8, PolarDB-IMCI implements two execution engines within each RO node. A row-based execution engine for point queries and a column-based execution engine for analytical queries. The optimizer of PolarDB-IMCI selects the appropriate execution engine for each query based on row-based cost estimation. By assuming that all queries can preferentially run in the row-based execution engine (i.e., low cost), the optimizer generates a row-oriented execution plan first. If the estimated cost of the row-oriented plan exceeds a threshold (i.e., high cost), a column-oriented plan will be generated and used over the column-based engine. The issue of intra-node routing is essentially a result of PolarDB using two executors. We leave the development of a new row-column hybrid cost model and hybrid execution as our future work.

6.2 IMCI Plan Generation

Instead of top-down constructing a column-oriented execution plan, PolarDB-IMCI transforms it from the row-oriented one. The transform workflow is shown in Figure 8. By doing so, column-oriented plans can preserve all behavioral characteristics.

For instance, in PolarDB-IMCI, implicit type casts of a column-oriented plan are always consistent with the row-oriented plan. During the plan generation, PolarDB-IMCI transforms the original expressions into a vectorized execution format to exploit SIMD instructions. This transformation is handled inside the expression objects (e.g., *Item* class in *MySQL*) and strictly follows up on original implicit type casts. Another instance is that column-oriented plans reuse error codes and messages from row-oriented plans. It is challenging to align errors across different execution engines. In PolarDB-IMCI, Column-oriented plans can retain static error detection directly from row-oriented ones to avoid this issue. For run-time errors, PolarDB-IMCI will fall back the execution to be row-oriented. As a result, PolarDB-IMCI achieves strong compatibility with the existing framework of *MySQL*.

Due to the differences between execution engines, column-oriented plans benefit less from following the join order of row plans. Thus, the optimizer further optimizes the join order after constructing a column-oriented plan. PolarDB-IMCI uses DPhyp [40] as the join ordering algorithm, which can efficiently handle various types of joins, including outer-joins and anti-joins. To provide accurate cardinality estimation for the optimizer, PolarDB-IMCI collects statistics through random

sampling. Sampling tasks will be performed periodically in the background. Furthermore, PolarDB-IMCI adaptively adopts various sampling methods [10, 11, 27] to make statistics efficient and accurate.

6.3 Execution Engine

To obtain advanced OLAP performance, PolarDB-IMCI designs a new high-performance analytical execution engine (i.e., column-based engine). Drawing on the knowledge of in-memory columnar databases [4, 25, 36], the analytical engine incorporates numerous state-of-the-art technologies, including a pipeline execution model, a set of well-optimized parallel operators, and a vectorized expression evaluation framework.

- **Pipeline Execution.** The execution tree of a vectorized execution plan is decomposed into multiple linear paths called pipelines. In a pipeline, a non-blocking operator (e.g., Filter, Join Probe) processes one batch at a time instead of all data, and then passes the intermediate result to the next operator. Pipeline execution brings several advantages: (1). a batch of data that streams through multiple operators is always cached; (2). intermediate results are reduced to minimize the memory footprint.
- **Parallel Operators.** To parallelize each pipeline, all operators in the analytical engine support parallel execution. For example, TableScan can concurrently fetch Data Packs in a non-interleaved manner, and the analytical engine implements Join as a lock-free partition Join [1]. Furthermore, to reduce cache misses, blocking operators use carefully designed data structures (e.g.,cache-friendly hash tables [3]) and software prefetching [13] as much as possible. Besides, all blocking operators have an optimized spill-to-disk version to handle out-of-memory crises, such as dynamic hybrid hash Join [30].
- **Expression Evaluation.** When a batch of data is cached, the performance bottleneck is switched from memory access to CPU computation. SIMD instructions, sometimes known as vectorized instructions, such as AVX-512, are powerful for accelerating CPU computation. Thus, an expression evaluation framework [38] is decoupled from operators to serve compute-intensive modules in a vectorized manner (i.e., using SIMD).

6.4 Strong Consistency

Since PolarDB-IMCI uses an asynchronous replication mechanism, analytical queries may observe stale data. For example, an analytical query may not read the updates that have already been committed in the RW node. However, it is possible for PolarDB-IMCI to achieve multiple consistency levels through the proxy layer to meet the requirement of applications, including strong consistency.

The proxy keeps track of the RW node's written LSN and all RO nodes' applied LSN. The written LSN and applied LSN indicate the transaction commit point for RW and RO, respectively. Transactions before the written LSN were committed on the RW node. Likewise, any log entries before the applied LSN are guaranteed to have been replayed by the RO node. The proxy may only route queries to the RO nodes whose applied LSN is not less than the written LSN to meet the requirements of strong consistency.

7 RESOURCE ELASTICITY

One of the core design concepts behind PolarDB-IMCI is to realize on-demand node provisioning with a storage-computation separation architecture. In this section, we dive into the node scale-out mechanism in PolarDB-IMCI.

Like most in-memory database systems [25, 31], PolarDB-IMCI periodically stores column indexes in shared storage as checkpoints to provide fast recovery after a system crash. More specifically,

in PolarDB-IMCI, new scale-out RO nodes can quickly construct their memory structures with checkpoints. In our implementation, the roles of RO nodes are divided into one leader and multiple followers. A leader is in charge of issuing checkpoints, while followers maintain their own memory structures, and leverage the checkpoints for fast recovery. The role assignment is centrally controlled by RW. When start-up, RW designates the first RO node in the cluster as an RO leader, and other RO nodes are followers. If the leader crashes, RW will re-designate one of the followers to be the new leader.

To take a checkpoint, the leader identifies the latest committed transaction sequence number as the Checkpoint Sequence Number (CSN). The transactions committed after the CSN are excluded in the checkpoint to enable a consistent data view between RO nodes. A major challenge is to ensure that the checkpointing tasks never stall the foreground log replay. However, checkpointing tasks may be stained when the log replay is in progress. Recall that there are three important in-memory structures (the RID locator, Packs, and VID maps) in RO nodes, and all of them should be coordinated with checkpoints. Addressing the challenge, PolarDB-IMCI handles each of them by the following steps.

- Packs in PolarDB-IMCI are append-only and immutable, which means the persistence timing of Packs is independent of checkpoints. Hence, Packs on the leader are written into PolarFS as soon as they are created. Visibility is controlled by VID maps.
- VID maps require a more careful design. Firstly, PolarDB-IMCI generates a copy of VID maps on the leader and parallelly checks all elements in the copy. If VIDs exceed the CSN, the elements in VID maps will be marked as invalid. Then, the visibility controlled by VID maps is aligned with the CSN and the copy can be persisted into PolarFS.
- RID locator splits a new immutable copy for checkpoints tasks by functional data structures [24]. Therefore, Subsequent transactions will not stain the checkpoint. Meanwhile, to prevent active transactions from leaving residues on the old view, checkpoints are only triggered when the memtable of the LSM tree is filled.

When adding a new RO node, PolarDB-IMCI first checks whether there is an available checkpoint of column indexes in PolarFS. If so, it loads the checkpoint and performs fast recovery; otherwise, it rebuilds column indexes from the row store. When starting from a checkpoint, RO nodes load the locator and VID maps into memory first. Regarding Packs, RO nodes use a lazy loading way. Only Packs accessed by queries are loaded into memory to reduce scale-out time. After that, the RO node replays the log entries after the checkpoint to catch up with the RO leader. During catching up, the RO node is able to serve queries with poor freshness. The poor freshness lasts only a short time. The experiments in §8.5 show that scaling out a RO node takes tens of seconds.

8 EVALUATION

8.1 Evaluation Setup

Configurations. The experimental evaluation was carried out on a PolarDB-IMCI cluster (mmx8.4 xlarge) in the Alibaba Cloud platform. Except for the scale-out experiment, we used two computation nodes, one read/write (RW) node and one read-only (RO) node. The scale-out experiment was conducted by adding RO nodes, thus consuming more than two nodes. The computation nodes are attached to a PolarFS volume which can provide nearly unlimited capacity. We used one ECS server (c7.8xlarge) on Alibaba Cloud as HTAP clients to issue SQL requests. The detailed configurations can be found in Table 1.

Benchmarks. To emulate diverse application scenarios and analyze the performance of PolarDB-IMCI systematically, we used three well-studied and widely-used benchmarks.

Table 1. Configurations of our evaluation.

RW/RO Node	32 vCPU, 1 NUMA node 256 *GB* DRAM
Client	32 vCPU 64 *GB* DRAM
OS	Alibaba Group Enterprise Linux Server release 7.2
Network	10Gbit/s Bandwidth
PolarFS	288000 IOPS (RandRead 16KB) 18000 IOPS (SeqWrite 128KB)

TPC-H [20] is adopted to evaluate the performance of PolarDB-IMCI in executing analytical queries. We used 100 *GB* and 1 *TB* of data volume, and reported the running time of each query. We also reported the geometric mean of all 22 queries, as suggested in the TPC-H official document.

We used CH-benCHmarks [17] to evaluate PolarDB-IMCI's performance under hybrid workloads. It integrates TPC-H queries into TPC-C [19] with a unified data schema. We reported the OLTP and OLAP throughput and studied the performance isolation property with a scale fact (i.e., the number of data warehouses) = 100.

To provide a more in-depth analysis of PolarDB-IMCI's micro component, sysbench [49] is used for pressure tests with diverse workload patterns. We set insert-only and write-only (update) workloads with Zipfian distribution. The database contains 100 tables using 64-bit integers as primary keys and 188 bytes per record.

We ran each experiment 10 times and reported the average number. Results were collected in the middle of each experiment to avoid the disturbance caused by system start-up and cool-down.

Our evaluation focused on the following questions:

§8.2 What is the overall performance of PolarDB-IMCI?
§8.4 Can PolarDB-IMCI achieve high data freshness?
§8.3 How does PolarDB-IMCI handle update propagation?
§8.5 Can PolarDB-IMCI achieve high resource elasticity when OLAP workloads increase?
§8.6 How does PolarDB-IMCI benefit real-world applications in production deployment?

8.2 Overall Performance

OLAP-only workloads. Achieving advanced OLAP performance (i.e., **G#2**) in an HTAP system is one of the foremost motivations of PolarDB-IMCI. In this evaluation, we compared the TPC-H query execution time of PolarDB-IMCI's column execution engine, its row execution engine (referred to as row-based PolarDB), and ClickHouse (an advanced OLAP system). For an apple-to-apple comparison, we built secondary indexes for each column in row-based PolarDB to maximize its performance. ClickHouse does not support PolarFS. To make a fair comparison with ClickHouse, PolarDB-IMCI was also configured to use local disks in this experiment. Currently, ClickHouse does not offer enough support for join reordering [16]. To evaluate the performance of execution engines separately, in the 1 *TB* experiment, we manually adjusted the join order of ClickHouse to the same as PolarDB-IMCI. Queries are executed one by one, and all systems used 32 threads for intra-query parallelism.

Figure 9 shows the results. With 100 *GB* data, PolarDB-IMCI achieved ×5.56 speed-ups (in geometric mean) compared to row-based PolarDB, and up to ×149.12 speed-ups for scan-intensive queries (e.g., *Q*10, *Q*15). With 1 *TB* data, the speed-ups are ×12.15 in geometric mean. The performance gain came from two folds: first, PolarDB-IMCI serves scan operations on column granularity, which

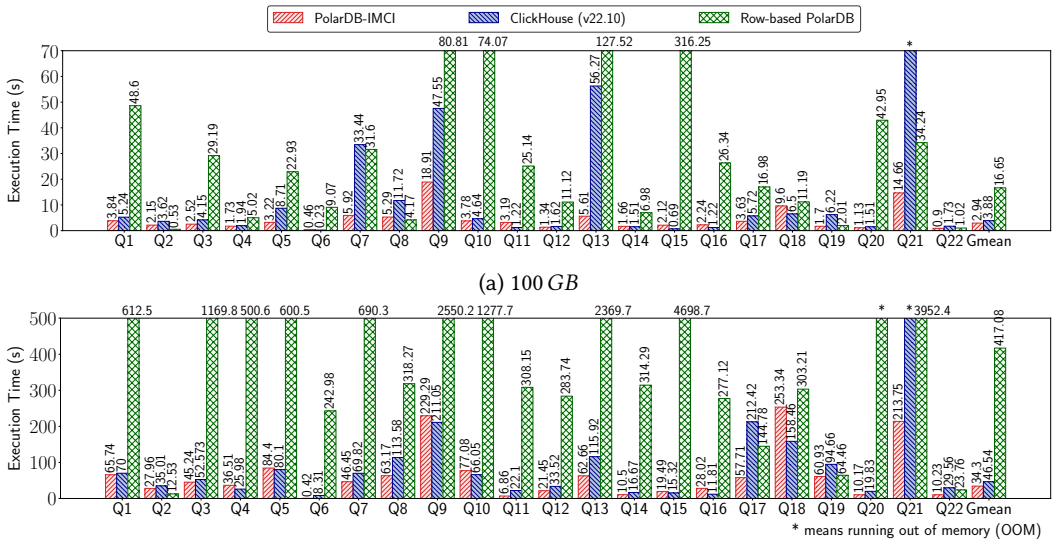


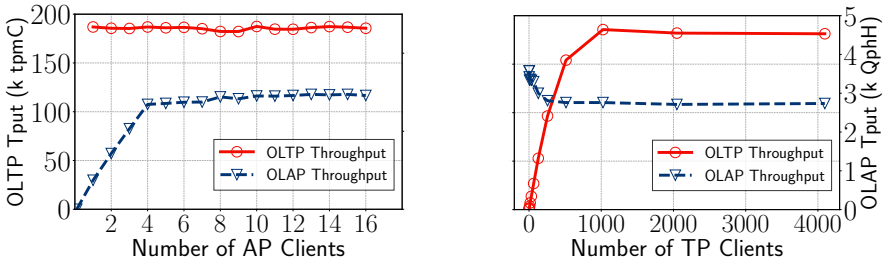
Fig. 9. Comparison of PolarDB-IMCI, PolarDB, and ClickHouse on TPC-H. All systems used 32 threads for intra-query parallelism.

minimizes read amplification caused by full table scan (§4); second, PolarDB-IMCI implements parallel operators, batch iteration, and SIMD optimizations to speed query processing on the large data volume (§6). One may find $Q2$ interesting: PolarDB-IMCI underperformed on such queries. This was because the selectivity of $Q2$ was low, and indexes built in row-based PolarDB were more efficient to handle point queries. However, thanks to our optimizer (§6.1), in practice, PolarDB-IMCI can automatically route such queries to their desirable execution engine.

Compared to ClickHouse, PolarDB-IMCI outperformed or was competitive with it on most queries. Overall, PolarDB-IMCI achieved ×1.32 speed-ups on 100 *GB* data and ×1.35 speed-ups on 1 *TB* data. PolarDB-IMCI incurred longer execution time on a small specific set of queries (e.g., $Q11$, $Q18$). To the best of our knowledge, this is caused by different implementation details in operators and memory management.

In summary, PolarDB-IMCI's OLAP performance is much better than row-based PolarDB and is comparable to ClickHouse.

HTAP workloads. We test PolarDB-IMCI's performance on hybrid workloads with CH-benCHmarks (§8.1). The results in Figure 10 show that PolarDB-IMCI has effective resource isolation. Following the standard [47], we evaluate PolarDB-IMCI in two rounds. First, we used 512 OLTP clients to saturate OLTP throughput (i.e., tune the number of clients to use 80% of CPU resources), and increased OLAP clients to issue analytical queries. As Figure 10a demonstrated, PolarDB-IMCI can perform at most 186890 tpmC (TPC-C NewOrder transactions per minute) and 2916 QphH (TPC-H query per hour) simultaneously, and the performance throughput degradation of OLTP is low (less than 1%). Second, we changed the roles of OLTP and OLAP, and let OLTP workloads increase after OLAP throughput was saturated. PolarDB-IMCI indeed incurred a little throughput degradation on OLAP throughput (<20%). We conclude this degradation for two reasons: (1). OLTP workloads enlarged the table size of some tables, thus higher OLTP throughput may degrade more OLAP performance; (2). the number of invalid rows in Packs increased with higher OLTP throughput. It validates our design choice of building column indexes on separated RO nodes.



(a) Perf. isolation of OLTP (b) Perf. isolation of OLAP

Fig. 10. Isolated OLTP and OLAP Performance of PolarDB-IMCI on CH-benCHmark Workloads. These two sub-figures share the same y-axis.

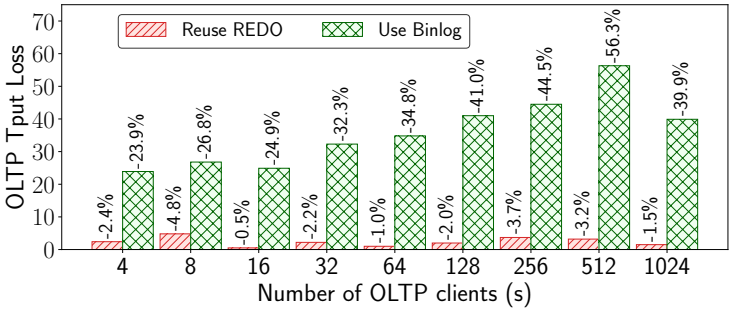


Fig. 11. Effectiveness of reusing REDO logs for updates propagation on sysbench write-only workload. The OLTP Throughput Loss is calculated by comparing PolarDB without IMCI.

We omit the comparison between PolarDB-IMCI and other transactional databases because the OLTP performance of PolarDB-IMCI strictly follows the performance of PolarDB [8, 9]. PolarDB-IMCI achieved good performance isolation between workloads (see Figure 10) and the overhead of enabling IMCI is low (see Figure 11).

8.3 Performance Perturbation

Then, we examine how the update propagation affects PolarDB's OLTP performance. Recall that minimal perturbation on OLTP (i.e., **G#3**) is pivotal to our consumers' experience. We design this experiment based on the sysbench insert-only workload, and calculated the throughput loss by comparing the throughput of candidate methods to the original throughput without IMCI (i.e., PolarDB with only row-based read-only replica). We started the experiment with an empty table and warmed up for 10 seconds. Figure 11 shows the results. Compared to using Binlog, PolarDB-IMCI's updates propagation methods (i.e., reusing REDO log) caused minimal performance perturbation to OLTP. The overhead of using Binlog was significantly higher because Binlog incurred additional fsyncs and more log IO. One may consider the drawback of reusing REDO is that PolarDB-IMCI has to parse physical logs to logical logs. However, it does not cause a bottleneck in log replay, as validated in our experiment (§8.4).

8.4 Data Freshness

Data freshness (i.e., **G#4**) is critical to the quality of analytical results. We evaluated data freshness by visibility delay (VD), which is the time taken for an update committed on an RW node to be

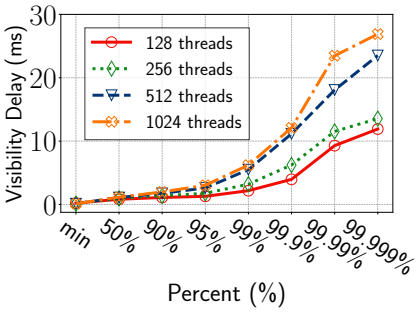


Fig. 12. VD on TPC-C.

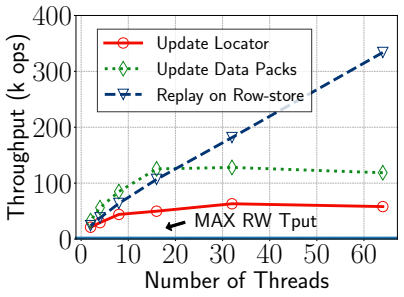


Fig. 13. Replay Performance.

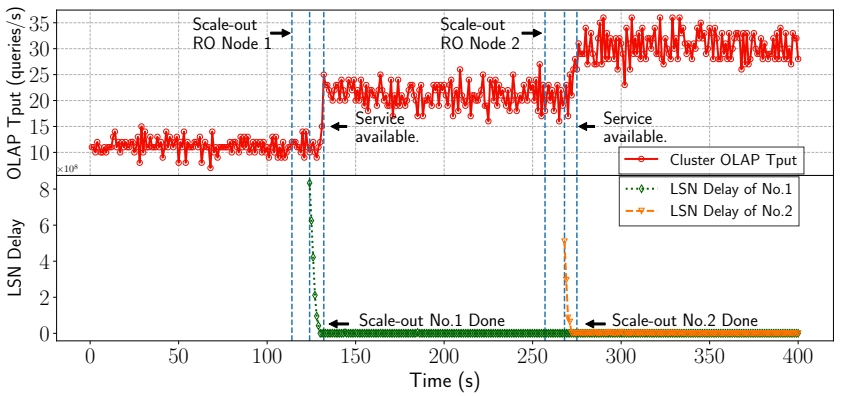


Fig. 14. Resource Elasticity on TPC-H

readable on RO nodes [12, 28]. Figure 12 provides the results of VD at different percentiles on TPC-C workloads with data warehouses = 100. PolarDB-IMCI achieved low visibility delay for three reasons: first, CALS (§5.1) minimized the update propagation window; second, the updates on column indexes are out-place and lightweight (§4.1); third, RDMA-equipped PolarFS reduced the shipping time.

Effectiveness of parallel replay. To provide additional support for the claim that components of column indexes should never be the bottleneck, we tested each component of PolarDB-IMCI individually, and report the maximum throughput on each component with varying threads. During the experiment, we used 512 OLTP clients to saturate OLTP throughput on the TPC-C workload and achieved 1934.97tps (i.e., 116098$tpmC$) throughput. Figure 13 shows the results. The maximum throughput of updating the RID locator and data Packs is much higher than the maximum throughput of OLTP on the RW node (×30.2 to ×61.3). Besides, replaying REDO logs on a row-based buffer pool is not the bottleneck. We also test the maximum throughput of physical log parsing (per thread) and committing. The throughputs are ~34k and ~459k respectively, which is also significantly higher.

8.5 Resource Elasticity

The desiderata on resource elasticity (i.e., **G#5**) drives our cloud-native implementation. To test the elasticity of PolarDB-IMCI, we used sysbench insert-only workloads with 3900 insertions per second (188 bytes per record) for the TP workload and TPC-H $Q6$ for the AP workload.

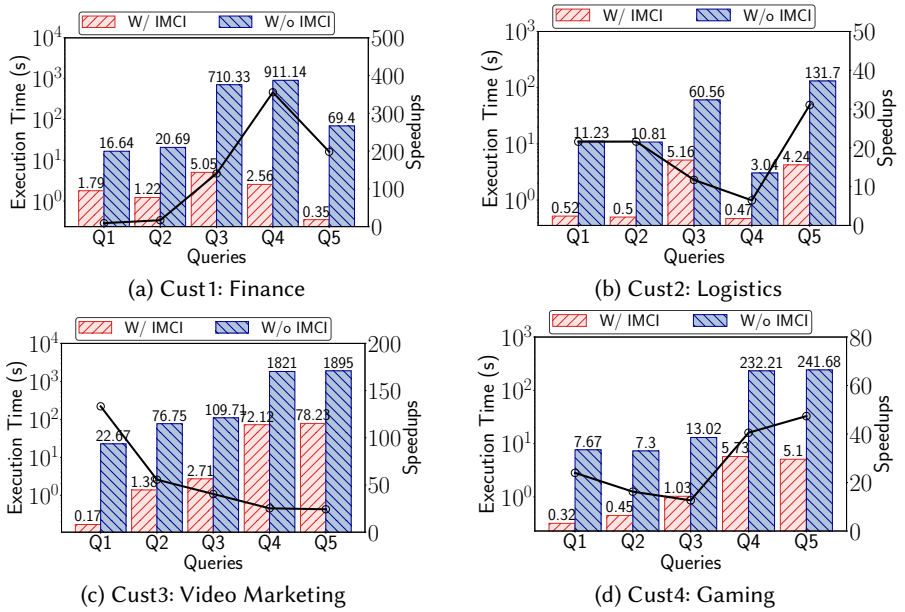
To scale out (i.e., add new RO nodes), PolarDB-IMCI relies on the checkpoints technique (§7) for a fast start-up. Figure 14 shows the results. We added the first new RO node into the cluster at

Table 2. Production workloads. The table describes different customer workload patterns.

Workload	DB Size	Tables	Max Table Size	Avg.# cols	Queries	Avg.# joins	Avg.# ops per plan
Cust1	2595.9 *GB*	997	393.3 *GB*	11.2	96	2.0	9.7
Cust2	163.2 *GB*	165	17.3 *GB*	27.2	311	1.3	10.0
Cust3	736.2 *GB*	681	91.5 *GB*	29.9	105	1.7	9.9
Cust4	47.8 *GB*	153	5.6 *GB*	13.5	106	9.0	41.9

Table 3. Distribution of queries at different IMCI speed-ups.

Speed-ups	Cust1	Cust2	Cust3	Cust4
[1, 2)	**55%**	**67%**	5%	0%
[2, 5)	12%	13%	5%	0%
[5, 10)	9%	5%	16%	1%
[10, 100)	23%	13%	28%	42%
[100, inf)	1%	2%	**46%**	**57%**



(a) Cust1: Finance

(b) Cust2: Logistics

(c) Cust3: Video Marketing

(d) Cust4: Gaming

Fig. 15. Speedups achieved by PolarDB-IMCI on representative queries. The left y-axis is in the log scale.

114s. It took 10s for PolarDB-IMCI to build in-memory components from the checkpoints. At 124s, when the newly added RO node (i.e., No.1) was able to serve the new incoming OLAP requests, the proxy server balanced the traffic and started new sessions to No.1. Thus, the cluster's OLAP throughput increased incrementally (see the top part of Figure 14). However, at the beginning of the start-up, the LSN delay of No.1 was extremely high (see the bottom part of Figure 14) since the new node still needed to catch up on updates committed after the checkpoint. Thanks to our high-performance updates propagation framework, No.1 could catch up to the latest state in a short time (9s). At 133s, No.1 could behave as a normal RO node to serve OLAP requests. We then added another new RO node (i.e., No.2) to the cluster at 257s. No.2 was able to provide services at 268s and could catch up to the latest at 276s. Notably, No.2 took less time to catch up to RW than

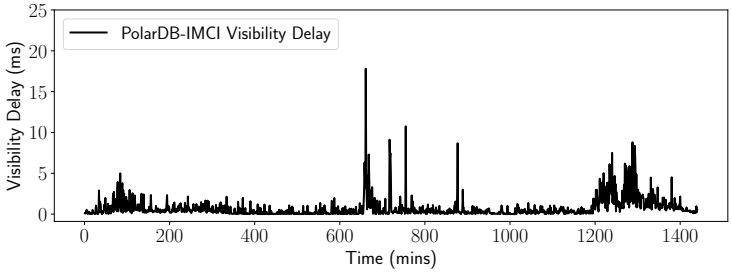


Fig. 16. Visibility Delay in real-world workloads.

No.1 since No.2 started from the next round checkpoint. Overall, PolarDB-IMCI achieved strong elasticity: it takes tens of seconds to scale out.

8.6 Performance of Production Deployment

In the last experiment, we studied PolarDB-IMCI's performance on several real-world customer workloads in a production environment. These workloads represent four diverse real-time applications where HTAP is highly desirable, i.e., finance, logistics, video marketing, and online gaming. Table 2 reports some aggregate statistics about the schema of these workloads. Generally, these customer workloads represent complex query patterns over diverse data schemas and database sizes. Table 3 shows the distribution of speed-ups achieved by PolarDB-IMCI compared to row-based PolarDB. Typically, PolarDB-IMCI can provide a faster speedup for more complex queries (i.e., those involving more joins and operations), such as Cust3 and Cust4. Additionally, Figure 15 shows the representative queries. We also calculated the speed-ups, which were shown on the right y-axis. In summary, the experimental results revealed that column indexes can result in orders of magnitude performance gains for slow SQL queries.

We then monitored the visibility delay between RW and RO nodes. The results are shown in Figure 16. During 24 hours, the visibility delay was changed with the customer's OLTP throughput and was always $<20ms$.

9 CONCLUSION

This paper present PolarDB-IMCI, a cloud-native HTAP database that achieves advanced OLAP performance with minimal perturbation on OLTP, and optimized visibility delay for better data freshness. PolarDB-IMCI adopts in-memory column indexes as complementary storage to speed up analytical queries, and introduced two key technologies for efficiency update propagation, i.e., CALS and 2P-COFFER. In addition, PolarDB-IMCI leverages checkpoints on shared storage to enhance fast computation resources scale-out. PolarDB-IMCI also absorbed excellent optimizations for complex query processing, and proposed a new query optimization flow. Our evaluation results show that PolarDB-IMCI can handle hybrid workloads efficiently in both experimental and productional environments.

ACKNOWLEDGMENTS

PolarDB-IMCI owes a great deal to our customers, whose feedback and suggestions were instrumental in the design of its architecture. We gratefully acknowledge the contributions of Ming Zhao, XuDong Wu, HuaWei Xue, and Shuai Jiang to the development of PolarDB-IMCI. Additionally, we extend heartfelt gratitude to the anonymous reviewers whose valuable comments greatly improved this paper.

REFERENCES

[1] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 362–373.

[2] Ronald Barber, Matt Huras, Guy Lohman, C Mohan, Rene Mueller, Fatma Özcan, Hamid Pirahesh, Vijayshankar Raman, Richard Sidle, Oleg Sidorkin, et al. 2016. Wildfire: Concurrent blazing data ingest and analytics. In *Proceedings of the 2016 International Conference on Management of Data*. 2077–2080.

[3] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-Efficient Hash Joins. *Proc. VLDB Endow.* 8, 4 (2014), 353–364.

[4] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*. www.cidrdb.org, 225–237.

[5] 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.

[6] Dennis Butterstein, Daniel Martin, Knut Stolze, Felix Beier, Jia Zhong, and Lingyun Wang. 2020. Replication at the speed of change: a fast, scalable replication solution for near real-time HTAP processing. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3245–3257.

[7] 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.

[8] 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.

[9] 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 '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 2477–2489.

[10] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. 2004. Effective Use of Block-Level Sampling in Statistics Estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*. ACM, 287–298.

[11] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1998. Random Sampling for Histogram Construction: How much is enough?. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. ACM Press, 436–447.

[12] Jianjun Chen, Yonghua Ding, Ye Liu, Fangshi Li, Li Zhang, Mingyi Zhang, Kui Wei, Lixun Cao, Dan Zou, Yang Liu, et al. 2022. ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3411–3424.

[13] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance through Prefetching. In *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*. IEEE Computer Society, 116–127.

[14] Zongzhi Chen, Xinjun Yang, Feifei Li, Xuntao Cheng, Qingda Hu, Zheyu Miao, Rongbiao Xie, Xiaofei Wu, Kang Wang, Zhao Song, et al. 2022. CloudJump: optimizing cloud databases for cloud storages. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3432–3444.

[15] Inc. ClickHouse. 2022. ClickHouse — open source distributed column-oriented DBMS. https://github.com/ClickHouse/ClickHouse/tree/22.6.

[16] Inc. ClickHouse. 2023. ClickHouse — Roadmap 2023. https://github.com/ClickHouse/ClickHouse/issues/44767.

[17] Richard L. Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi A. Kuno, Raghunath Othayoth Nambiar, Thomas Neumann, Meikel Poess, Kai-Uwe Sattler, Michael Seibold, Eric Simon, and Florian Waas. 2011. The mixed workload CH-benCHmark. In *Proceedings of the Fourth International Workshop on Testing Database Systems, DBTest 2011, Athens, Greece, June 13, 2011*. ACM, 8.

[18] Apache Community. 2023. Apache Flink. https://flink.apache.org/.

[19] THE TRANSACTION PROCESSING COUNCIL. 2014. TPC-C. http://www.tpc.org/tpcc/.

[20] THE TRANSACTION PROCESSING COUNCIL. 2023. TPC-H. http://www.tpc.org/tpch/.

[21] 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 *Proceedings*

of the 2016 International Conference on Management of Data. 215–226.

[22] 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.

[23] Alex Depoutovitch, Chong Chen, Jin Chen, Paul Larson, Shu Lin, Jack Ng, Wenlin Cui, Qiang Liu, Wei Huang, Yong Xiao, et al. 2020. Taurus database: How to be fast, available, and frugal in the cloud. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1463–1478.

[24] James R Driscoll, Neil Sarnak, Daniel D Sleator, and Robert E Tarjan. 1989. Making data structures persistent. *Journal of computer and system sciences* 38, 1 (1989), 86–124.

[25] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.

[26] 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 *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1917–1923.

[27] Peter J Haas and Lynne Stokes. 1998. Estimating the number of classes in a finite population. *J. Amer. Statist. Assoc.* 93, 444 (1998), 1475–1487.

[28] 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.

[29] 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 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery, 651–665.

[30] Shiva Jahangiri, Michael J. Carey, and Johann-Christoph Freytag. 2022. Design Trade-offs for a Robust Dynamic Hybrid Hash Join. *Proc. VLDB Endow.* 15, 10 (2022), 2257–2269.

[31] Tirthankar Lahiri, Shasank Chavan, Maria Colgan, Dinesh Das, Amit Ganesh, Mike Gleeson, Sanket Hase, Allison Holloway, Jesse Kamp, Teck-Hua Lee, et al. 2015. Oracle database in-memory: A dual format in-memory database. In *2015 IEEE 31st International Conference on Data Engineering*. IEEE, 1253–1258.

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

[33] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1740–1751.

[34] Per-Åke Larson, Cipri Clinciu, Eric N Hanson, Artem Oks, Susan L Price, Srikumar Rangarajan, Aleksandras Surna, and Qingqing Zhou. 2011. SQL server column store indexes. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 1177–1184.

[35] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[36] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 743–754.

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

[38] Meng Li, Zheyu Miao, Di Wu, Feifei Li, Sheng Wang, Wei Cao, Zhi Qiao, Yubin Ruan, Yukun Liang, Jimmy Yang, Haipeng Dai, and Guihai Chen. 2023. ROVEC: Runtime Optimization of Vectorized Expression Evaluation for Column Store. *IEEE Trans. Knowl. Data Eng.* 35, 3 (2023), 3045–3058.

[39] 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.

[40] Guido Moerkotte and Thomas Neumann. 2008. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. ACM, 539–552.

[41] MySQL. 2019. MySQL 8.0.18 (2019-10-14, General Availability). https://dev.mysql.com/doc/relnotes/mysql/8.0/en/news-8-0-18.html.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[43] Oracle. 2018. Database-Level Supplemental Logging. https://docs.oracle.com/database/121/SUTIL/GUID-D2DDD67C-E1CC-45A6-A2A7-198E4C142FA3.htm.

[44] Sukhada Pendse, Vasudha Krishnaswamy, Kartik Kulkarni, Yunrui Li, Tirthankar Lahiri, Vivekanandhan Raja, Jing Zheng, Mahesh Girkar, and Akshay Kulkarni. 2020. Oracle database in-memory on active data guard: Real-time analytics on a standby database. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1570–1578.

[45] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric N. Hanson, Robert Walzer, Rodrigo Gomes, and Nikita Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 2340–2352.

[46] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M Lohman, et al. 2013. DB2 with BLU acceleration: So much more than just a column store. *Proceedings of the VLDB Endowment* 6, 11 (2013), 1080–1091.

[47] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 219–238.

[48] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012*. ACM, 731–742.

[49] SysBench. 2023. SysBench. https://github.com/akopytov/sysbench.

[50] Ben Vandiver, Shreya Prasad, Pratibha Rana, Eden Zik, Amin Saeidi, Pratyush Parimal, Styliani Pantela, and Jaimin Dave. 2018. Eon Mode: Bringing the Vertica Columnar Database to the Cloud. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 797–809.

[51] Panos Vassiliadis. 2009. A survey of extract–transform–load technology. *International Journal of Data Warehousing and Mining (IJDWM)* 5, 3 (2009), 1–27.

[52] 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.

[53] 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 *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1041–1052.

[54] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, et al. 2018. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*. 789–796.

[55] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, et al. 2020. F1 Lightning: HTAP as a Service. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3313–3325.

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

[57] 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.

Received November 2022; revised February 2023; accepted March 2023