Vodka: Rethink Benchmarking Philosophy in HTAP Systems

ABSTRACT

For real-time analysis of up-to-date data, hybrid transaction/analytical processing (HTAP) systems have been extensively studied. Three techniques play a critical role in the HTAP systems, which are resource isolation (reduce resource contentions), consistency model (read an appropriate data version), and data sharing (transfer data swiftly). However, there still lacks a benchmark suite that comprehensively covers these techniques. The core challenges come from the requirements of: (a) consistent workload resource consumption (provide workloads with the same computational complexity); (b) query-oriented freshness evaluation (focus on the degree of version staleness in the range of queried data); (c) precise data sharing efficiency measurement (catch the synchronization status accurately). For resource isolation, we formalize the query cardinality changes under dynamic modifications, and manipulate the cardinalities of various query operators to ensure consistent query complexity comparisons under any data size. For consistency model, we design a fine-grained version management strategy based on which query-oriented freshness is calculated. For data sharing, we design a lightweight point query driven method to check the synchronization status accurately. We finally conduct extensive experiments on three representative systems to justify our designs and provide insights for future development.

1 INTRODUCTION

With the increasing demand for real-time analytics, such as fraud detection [45] and stock price monitoring [51], many database vendors [7, 15, 22, 24, 28, 30, 50, 73, 74] propose to combine OLTP and OLAP engines to construct a new type of system named HTAP system [43]. In general, the OLTP engine (denoted as T-engine) is designed to capture, store, and process data from transactions in real-time, and it's optimized for row-oriented operations [1, 49]. In contrast, the OLAP engine (denoted as A-engine) is optimized for conducting complex data analysis on large amounts of historical data for smarter decision making, and the data is generally organized by column-based layout in a highly compressed form [25, 72]. In addition to the design targets mentioned above, the HTAP systems also focus on the optimization of three key techniques: resource isolation, consistency model, and data sharing.

As illustrated in Fig. 1, the HTAP system tries to connect data generation and consumption in a real-time manner. The resource isolation technique is employed to schedule and separate hardware resources such that resource contentions between T- and A-engines are effectively mitigated, which further ensures operational stability in both the data generation and consumption phases. The data sharing technique helps efficiently transfer the produced fresh data from the T-engine to the A-engine for consumption. Moreover, as data synchronization is practically done asynchronously to make better performance isolation between OLTP and OLAP workloads, the HTAP system uses the consistency model to specify the A-engine to read appropriate data versions generated by the T-engine. However, the quality of all three techniques cannot be concurrently guaranteed in any HTAP system. For example, placing the two engines on separate nodes with hardware isolations could absolutely eliminate resource interferences, but complicate the efficiency of data sharing and might further decrease data freshness [22]. As a result, existing HTAP systems propose to strike a trade-off between the design decisions of the three techniques [32].

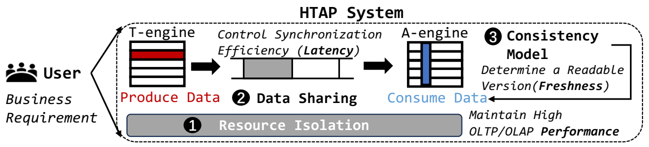


Figure 1: Three Key Techniques in HTAP Systems

Recently, HTAP benchmarks have been extensively studied in the literature [55]. However, providing a comprehensive evaluation of the three critical techniques in HTAP systems has always been a tough work. The core challenges come from the requirements of:
Consistent Workload Resource Consumptions (C1). The resource isolation technique aims to ensure that integrating OLTP and OLAP businesses into one system does not cause evident performance interferences or degradations [46]. This involves not only meeting strict requirements of stable OLTP throughput (TPS) according to the specific business scenario, but also achieving adequate queries per hour (QphH) [37, 42, 55]. Previous benchmark studies propose to use heuristic methods [12, 39] to combine the two performance metrics, such as $\sqrt{TPS \times QphH}$. However, they do not consider the requirement of consistent computational complexities and resource demands when running the workload on different HTAP systems. This is because different systems might have different OLTP throughputs, which leads to different increasing rates of data sizes in both T- and A-engines. Then, inconsistent data sizes would result in inconsistent scan sizes and cardinality of each operator in OLAP queries, both of which determine the query complexity and resource consumptions [34]. Therefore, it's not a fair comparison between different HTAP systems when they face workloads with inconsistent computational complexities.
Query-oriented Freshness Evaluation (C2). The consistency model determines the gap in data versions between the T- and A-engines, and its effectiveness is typically evaluated using the data freshness metric. Previous work [39] mainly considers the global system consistency status, and measures freshness as the gap between the query start time and the earliest commit time of transactions that are invisible to the A-engine. However, users are usually concerned whether their queried data is sufficiently fresh such that their specific application requirements are satisfied [6, 9]. It is acceptable even if the data outside the accessed range is stale. Unfortunately, there exists no method that could flexibly measure the freshness of data accessed by each query.
Precise Data Sharing Efficiency Measurement (C3). Data sharing efficiency not only determines the freshness of data in the A-engine, but also affects the query response times under some specific consistency models. More specifically, for HTAP systems with

a high consistency model, queries usually wait for the synchronization to finish before returning results, and their response times are constrained by the synchronization efficiency [15, 22, 74]. Although HTAP systems that tolerate a certain degree of data freshness loss do not force queries to wait for the synchronization procedure to complete [7, 73], they still strive to optimize synchronization efficiency to improve freshness as much as possible. However, the precise measurement of synchronization latency is still vacant in existing researches. The main challenge is that the real-time synchronization status of the system is invisible in a black-box mode. Moreover, instrumenting the kernels to catch the internal synchronization state of HTAP systems is laborious or even impossible, especially for closed-source or cloud-native databases.

We propose *Vodka* to address the above challenges. To address **C1**, we propose to regulate the instantiation of OLAP query parameters according to the scale of underlying tables, which ensures that the complexity of each OLAP query can be effectively derived according to the table size (i.e., data size). Specifically, we first formalize the cardinality change of different types of query operators under dynamic modifications. Based on this formation, we leverage *set transforming rules* to reduce the complex predicates into simple cases, and then propose data distribution-guided methods to manipulate the data updates and query parameter instantiations such that the cardinality of each query operator increases linearly with the scale of data size. Since data size and query cardinality have a critical impact on query complexity, we further design a mathematical regression model to estimate the relationship between data size and query latency for each *A*-engine. With this mechanism, we can compare the query performance between different *A*-engines even though the queries are executed under different data sizes.

To address **C2**, we propose a new definition of freshness for different consistency models, which specifies the differences between the latest data version written in the *T*-engine and the data version being queried in the *A*-engine. The ideal method is to compare the data items touched by the query in the *A*-engine with their corresponding MVCC version chains inside the *T*-engine. However, this internal structure is invisible to users. Then, we propose a lightweight and decoupled version tracking approach. Specifically, we attach a version column in each table to track the evolution of the tuple-level version on the server side, which helps to identify the data version touched by the query at the granularity of tuples. Moreover, we leave the expensive track of each column value's version evolution on the client side, maintaining the actual evolution of each column value along with the evolution of its corresponding tuple. This is because the evolution of a tuple does not indicate the change of a specific column value. In this way, we can effectively reduce performance interferences in the *T*- and *A*-engines. Additionally, we also design some garbage collection methods to remove unnecessary data versions to reduce memory usage.

To address **C3**, we propose a point query driven method which only tracks the synchronization state of the last committed write before the test time point. This is because the order of synchronized writes is consistent with the commit order of writes in the *T*-engine [4, 31, 71]. Then, we can guarantee that all the writes before the test point have been successfully synchronized if the last committed write is visible in the *A*-engine. To this end, on the client side, we monitor the commit timestamp of each write before the test point and maintain a variable to track the write with the largest commit timestamp. Then, we repeatedly launch a lightweight point query to retrieve the last committed write until it's visible in the *A*-engine. Since the response time of point query is quite low, it ensures that synchronization completion time is caught in time.

In summary, we make the following contributions:

(1) We are the first approach to propose a comprehensive HTAP benchmarking philosophy addressing the challenges C1-C3.
(2) We implement an open-sourced benchmark suite [63] following the above philosophy.
(3) We launch extensive experiments over three typical systems with our benchmarking suite to provide novel insights for future HTAP system development.

2 BACKGROUND

2.1 HTAP Key Techniques

To support efficient transaction processing while ensuring effective real-time analytics in HTAP systems, three key techniques are widely adopted [9, 22, 28, 35, 50, 51, 74].

2.1.1 *Resource Isolation.*

HTAP systems employ resource isolation strategies to schedule and separate the hardware resources for mitigating resource contentions between the hybrid workloads and maintaining high performance of *T*- and *A*-engines [46]. In general, there exist two kinds of strategies, which are *unified resource* and *decoupled resource.*

Unified Resource. Both *T*- and *A*-engines share all resources of the same node, and they use logical isolation methods to schedule resources [46]. Specifically, for the CPU resources, the virtualization technique can be applied to split resources into groups and assign them to OLTP and OLAP workloads separately according to their predefined usage limits [35, 74]. Another way is to bind CPUs to different NUMA nodes to mitigate contentions [37, 58]. For memory resources, the virtualization technique can also be applied [35], or maintaining two in-memory data replicas to serve OLAP reads and OLTP writes, respectively [15, 17, 28].

Decoupled Resource. The *T*- and *A*-engines are deployed on different nodes, and *memory isolation* and *disk isolation* methods are usually applied to achieve different kinds of resource isolations. On the one hand, for memory isolation, each of the *T*- and *A*-engines could use all the CPU and memory resources of their assigned nodes [7, 61], but they need to share all underlying storage resources. On the other hand, for disk isolation, *T*- and *A*-engines have their dedicated CPU, memory, and storage resources through absolute hardware isolations [22, 73].

In summary, the unified resource strategy performs logical resource isolations, which is less effective in reducing resource contentions than the decoupled resource strategy since they are still under the same physical environment. As a result, it could not absolutely eliminate resource interferences, especially for disk IO and network bandwidth [35]. However, it enables more efficient resource utilization within each node since the decoupled resource strategy might lead to lots of idle resources.

2.1.2 *Consistency Model.*

As data transfer delays between *T*- and *A*-engines might occur, there exist version discrepancies between data records in *T*- and *A*-engines, which further results in various

degrees of data freshness. In general, the specific version gap is regulated by the consistency model, which can be broadly classified as *linear consistency*, *sequential consistency*, and *session consistency*.
Linear Consistency. It requires the data accessed on *A*-engine must be exactly the same as the latest version on *T*-engine, which is widely adopted by Oracle, SQL Server, OceanBase, and TiDB.
Sequential Consistency. It allows the *A*-engine to access the previous snapshot of the *T*-engine, but all *A*-clients should read the consistent snapshot versions at the same time point. ByteHTAP, BatchDB, and Vegito are the representative systems.
Session Consistency. It allows each client of the *A*-engine to read its snapshot versions from the *T*-engine individually, which is adopted by PolarDB, PostgreSQL Streaming Replication (PG-SR), Aurora, and Hyper.

In summary, the data freshness is fundamentally determined by the consistency model, which regulates whether the *A*-engine should select a specific snapshot of the *T*-engine for reading. As different consistency models impose different synchronization pressures on the HTAP system [51], they can be used to balance performance and freshness according to the requirements of the business.

2.1.3 Data Sharing. It refers to the method of sharing the produced fresh data from the *T*-engine to *A*-engine for consumption [54]. In general, there exist three kinds of data sharing methods, which are *share everything*, *share storage*, and *share nothing*.
Share Everything. The *T*- and *A*-engines share a single copy of data in the storage layer. With the help of MVCC mechanism [70], it allows the *A*-engine to access multi-versions of records created by the *T*-engine. Although this method avoids the high cost of data copying or synchronization, the *A*-engine might need high overhead in traversing version chains to find the expected versions. Moreover, if OLAP queries hold old-version records for a long time, garbage collection for stale versions will be hard [26].
Share Storage. The *T*- and *A*-engines share the same storage layer but maintain two copies of data in memory. One approach is based on the snapshot mechanism [24]. Specifically, when each query arrives, it first triggers system-level APIs to create a virtual in-memory data snapshot from the most recent version for subsequent reading on demand. Note that, this snapshot is initially not physically separated from the basic data. For new OLTP updates, the system employs a Copy-on-Write (CoW) mechanism, where any modification on the data included in the snapshot would lead to the modified data being copied to a new memory location. Thus, it results in high memory usage in write-intensive scenarios, since we need to allocate large memory space for frequent updates [32]. Another approach proposes to maintain an additional in-memory column store for the *A*-engine. The updates are first performed on the *T*-engine's row store. Then, when a certain amount of delta updates are accumulated, they will be merged into the column store in batch. However, maintaining two copies of data in memory would lead to a huge amount of memory consumption [15].
Share Nothing. The system maintains two copies of data in separate storage layers, where a transactional copy serves OLTP workloads and an analytical copy serves OLAP workloads. The updates in the transactional copy are asynchronously transferred to the *A*-engine by packing them as logs, which are further reorganized into a column layout. However, this results in significant cost in log shipping and replaying [22, 73].

In summary, *share everything* avoids the expensive data synchronization compared with the *share nothing*, offering high data sharing efficiency. However, it struggles to apply general column store optimizations on *A*-engine such as data compression and vectorized execution. This is because its MVCC-based data store is generally optimized for the *T*-engine [1]. The method of *share storage* serves as a compromise between data sharing efficiency and OLAP optimizations. Based on the techniques described above, Fig. 2 shows a classification of popular HTAP systems.

2.2 Observation From Existing Benchmarks

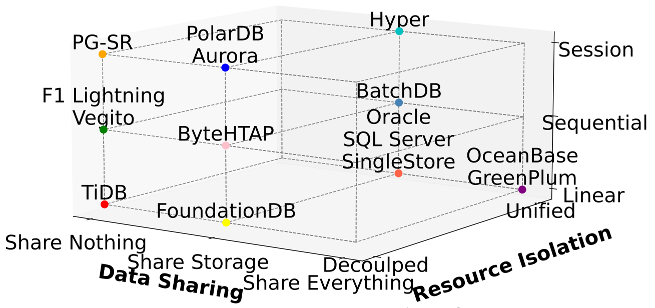


Figure 2: HTAP Systems Classification

Evaluate Resource Isolation. Some studies propose to compare the effectiveness of resource isolation among different HTAP systems by using heuristic methods [12] to combine the two performance metrics (e.g., $\sqrt{TPS * QphH}$). However, they overlook the fact that the computational complexity of OLTP and OLAP workloads is quite different, and different systems might face workloads with inconsistent computational complexities and resource demands. This is because different systems might exhibit different OLTP throughputs, which further leads to an inconsistent data size growth rate. However, different data sizes would also lead to different scan sizes and cardinality of each operator in OLAP queries, both of which have a critical impact on the query complexity [34]. Moreover, even under the same OLTP throughput with stable data size growth, the start time of each system's OLAP query might differ a lot since they have different OLAP processing capabilities. As data size and data distribution usually dynamically change over time, this implies that the cardinality of each query operator would change indeterminately, leading to inconsistent query complexities. Therefore, the calculation of *QphH* based on the average latency of queries is not fair since the query complexity at different timepoints is unknown. To address this, previous studies [11, 12, 23, 39] propose to restrict queries to accessing attributes with unchanged distributions. However, this restriction cannot meet the demands of real-world business scenarios. Furthermore, these studies usually fix query parameters. But even under fixed query parameters, different query start times would lead to different data sizes and cardinalities of each operator, causing different query complexities.
Evaluate Consistency Model. The consistency model describes the differences in data versions visible to the *T*- and *A*- engines,

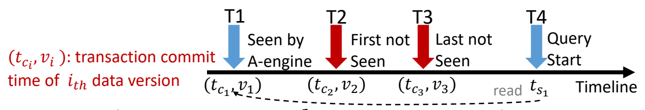


Figure 3: Data Freshness Demonstration

which is also widely known as the degree of data freshness. However, most of the existing work lacks an effective method for freshness calculations [39]. Although *HATtrick* [39] proposed to define the freshness as the gap between the query start time t_{s_1} in A-engine and the earliest commit time t_{c_2} of T-engine's transactions that are invisible to the A-engine (denoted as t_{s_1}-t_{c_2} in Fig. 3). However, according to the concept of freshness, it has two drawbacks. Firstly, users are more concerned with whether they could visit the most recent version of data within the query range [6, 62]. Thus, the freshness calculation method should not merely consider the global system consistency status, but rather focus on the query's specific access range. This is because data outside the query range does not contribute to online analytics and its staleness is imperceptible to the user experience. Secondly, the query start time is not suitable for freshness calculation, since it fails to reveal the concrete degree of data staleness for user-visible data in the A-engine compared to the freshest data in the T-engine. For example, consider the record in Fig. 3, the T-engine maintains version 3 (v_3) of the record, while the A-engine maintains version 1 (v_1) whose commit time is 1 second ahead of v_3. Then, an hour later at t_{s_1}, suppose a query initiated on the A-engine could still only access v_1 of the record. If the freshness is calculated based on the query start time, it would indicate the data is extremely stale. However, even if an analytical query is performed on the T-engine, the data being analyzed would only be 1 second later than what is observed in the A-engine, which is unlikely to cause a significant analytical bias. In this case, it only indicates that the HTAP system has poor data sharing capabilities and data is not synchronized promptly, rather than reflecting a true staleness. So, the freshness should be calculated as t_{c_3}-t_{c_1}=$1s$.

Evaluate Data Sharing. Data sharing determines the efficiency of data transfer (i.e., synchronization) between the T- and A- engines. In general, HTAP systems with a high consistency model require synchronization to finish before responding to users, making the optimization of data sharing efficiency crucial for improving query latencies. In contrast, systems with a weaker consistency model could return query results without waiting for the synchronization procedure to finish. However, they still invest significant time and effort to improve data freshness [7]. Therefore, we need to accurately assess the efficiency of data sharing strategies between different systems. Moreover, since users are more concerned with whether the data written before a certain time point have been fully synchronized to the A-engine [9, 51, 64] to perform subsequent analytic tasks, we propose to benchmark the data synchronization latency in various time points in a black-box mode. This is because instrumenting the kernels to catch the internal synchronization status of HTAP systems is laborious or even impossible. Unfortunately, none of the existing studies has proposed methods for evaluating the data sharing capabilities of various HTAP systems.

3 VODKA FRAMEWORK

The framework of *Vodka* (in Fig. 4) includes three components to benchmark resource isolation, consistency model, and data sharing.

Resource Isolation. The OLTP business usually plays a mission-critical role in real-world applications, and users tend to have strict requirements of stable OLTP throughput according to the specific business scenario [37, 42, 55]. That is, users are primarily concerned about whether the HTAP system could mitigate resource contentions well, which ensures that the system could achieve optimal OLAP performance while meeting the requirement of a specific OLTP throughput [37, 42, 55]. Since consistent computational complexity leads to the same resource consumption pressure, to ensure a fair comparison, we propose to follow the paradigm of comparing the performance of A-engine under the same OLTP throughput [11, 50]. However, the query complexity in the A-engine is still inconsistent due to different query start times and data sizes (see § 2.2). To address this issue, this component consists of three key steps: 1) formalize the cardinality change of different query operators under dynamic modifications; 2) use the formalization to guide the data manipulations in the T-engine and query parameter instantiations in the A-engine, which ensures that the cardinality of query operators increases linearly with the scale of data size; 3) construct a regression model to deduce the query latency of each A-engine according to the underlying data size, which enables us to compare the query performance of different A-engines even if the queries are performed under different data sizes.

Consistency Model. We compute data freshness based on the differences in commit time between the latest data version written in the T-engine and the data version being queried in the A-engine. Though the internal MVCC version chain provides an opportunity to trace the data versions, it is generally inaccessible to users and cannot distinguish the data versions with the same value. Then we propose a lightweight and decoupled version tracking approach to trace version evolutions. The main idea is to attach a new version column to each table to track the tuple-level version evolution on the server side. The detailed column data evolution time along with the column-level values involved in the evolution, are stored on the client side. Finally, the freshness exposes the maximum commit timestamp gap between the T- and A-engines regarding all the query accessed data items.

Data Sharing. To effectively measure synchronization latency in black-box systems where the internal synchronization state is not visible, we introduce a point query driven method to avoid laborious invasion of systems. Specifically, we track the last committed write with the largest witnessed timestamps on the client side. Due to the linearizability of writes in HTAP systems, the synchronization status of this last committed write guarantees that all prior writes are synchronized. Similarly, we attach a version column to each table to indicate the version evolution for the latest write, which helps monitor whether the A-engine has achieved synchronization. We conduct a point query to check the version consistency of this last committed write, repeating this until the consistency is confirmed. The time taken to achieve this consistency indicates the data sharing latency. Considering the rapid response of the point query, the synchronization completion can be captured in time.

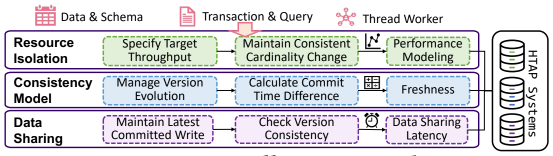


Figure 4: *Vodka* Framework

4 BENCHMARK RESOURCE ISOLATION

In this section, we first formalize the change of query cardinality under dynamic modifications (§ 4.1). Then we explore how to manipulate the data updates and query parameter instantiation such that the cardinality of each operator increases linearly with the scale of data size, including selection (§ 4.2), join (§ 4.3), and other operators (§ 4.4). Since query cardinality has a critical impact on query performance, we propose a mathematical model to capture the relationship between cardinality and query latency (§ 4.5). Taken together, it enables us to achieve a fair comparison of systems with different OLAP capacities under the same query complexity.

4.1 Query Cardinality Formalization

In this section, we first introduce some basic concepts of query tree and cardinality. Consider a query q, its logical execution plan can be represented by a query tree. Each leaf node in the query tree represents a table R_i involved in q. Each table R_i has one primary key column $R_i.PK$, several non-key attribute columns $R_i.A_1, ..., R_i.A_m$, and zero or more foreign key columns $R_i.FK_1, ..., R_i.FK_n$. We use $\sigma_P(R_i)$ to denote a selection operation with a predicate P on table R_i, and use $R_i \bowtie_J R_j$ to denote a join operation between two tables R_i and R_j with a join predicate J. The projection and aggregation are denoted as Π and $\mathcal{G}$. The selectivity $Sel(\sigma_P(R_i))$ of a selection operator is defined as the proportion of tuples in table R_i that satisfy the predicate P. It can also be viewed as the probability of event $\sigma_P(R_i)$ occurring in the value space of R_i, i.e., $Pr(\sigma_P(R_i))$ [16]. Thus, the cardinality of $\sigma_P(R_i)$ can be computed as $Card(\sigma_P(R_i))=Sel(\sigma_P(R_i)) \cdot |R_i|$. Similarly, the selectivity of a join operator is denoted as $Sel(\sigma_P(R_i) \bowtie_J \sigma_P(R_j))$, which can be viewed as the probability of an event where both the selection and join predicates are satisfied simultaneously in the value space $R_i \times R_j$. That is, $Sel(\sigma_P(R_i) \bowtie_J \sigma_P(R_j)) = Pr(\sigma_P(R_i), \sigma_P(R_j), R_i \bowtie_J R_j)$.

However, in real-world scenarios, there might exist intricate dependencies between columns. Although cardinality estimation has been extensively studied, it's still challenging to derive accurate cardinality under arbitrary query templates and parameters [27]. In light of this, we propose to follow previous OLAP benchmarks [41, 59, 60] and introduce two independence principles to decouple complex data dependencies during the data generation phase. Specifically, the first is the independence of selection results between any two tables R_i and R_j. It indicates that the non-key columns of R_i are independent of the non-key columns of R_j (see Eq. 1). Therefore, the selection result on one table does not affect the selectivity of any selection on another table. The second is the independency between join and selection results, which indicates that the non-key columns are independent of the key columns in each table. That is, for any selection operator σ_P, the selectivity of σ_P on a single table R_i is the same as that of applying σ_P on the join result of two tables R_i and R_j (see Eq. 2).

$$
\begin{aligned}
Pr(\sigma_P(R_i)|\sigma_P(R_j))&=Pr(\sigma_P(R_i)),\\
Pr(\sigma_P(R_j)|\sigma_P(R_i))&=Pr(\sigma_P(R_j)) \qquad (1)\\
Pr(\sigma_P(R_i)|R_i \bowtie_J R_j)&=Pr(\sigma_P(R_i)),\\
Pr(\sigma_P(R_j)|R_i \bowtie_J R_j)&=Pr(\sigma_P(R_j)) \qquad (2)
\end{aligned}
$$

In addition, based on the Bayes theorem [67], the selectivity in Eq. 3 can be converted to Eq. 4. Moreover, according to the two independence principles described above, we can further decouple it into the product of three independent probabilities (see Eq. 5).

$$
\begin{aligned}
Sel(\sigma_P(R_i) \bowtie_J \sigma_P(R_j)) &= Pr(\sigma_P(R_i), \sigma_P(R_j), R_i \bowtie_J R_j) \qquad (3)\\
&= Pr(\sigma_P(R_i), \sigma_P(R_j)|R_i \bowtie_J R_j) \cdot Pr(R_i \bowtie_J R_j) \qquad (4)\\
&= Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot Pr(R_i \bowtie_J R_j) \qquad (5)
\end{aligned}
$$

Next, the cardinality of the combination of join and selection operators can be calculated by Eq. 6, where $|R_i|$ and $|R_j|$ denote the table size. Lastly, as the T-engine would serve data writes from clients, the data in each table R_i is continuously evolving along with time, i.e., R_i becomes R_i' after Δ_i updates. Then, the cardinality change of $\sigma_P(R_i) \bowtie_J \sigma_P(R_j)$ can be formalized as Eq. 7.

$$
\begin{aligned}
&Card(\sigma_P(R_i) \bowtie_J \sigma_P(R_j))\\
&=Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot Pr(R_i \bowtie_J R_j) \cdot |R_i| \cdot |R_j|\\
&=Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot |R_i \bowtie_J R_j| \qquad (6)\\
&Card(\sigma_P(R_i) \bowtie_J \sigma_P(R_j))\\
&=Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot |R_i' \bowtie_J R_j'| \qquad (7)\\
&s.t. \quad R_i'=R_i \cup \Delta R_i \text{ and } R_j'=R_j \cup \Delta R_j
\end{aligned}
$$

4.2 Selection Selectivity Control

The core idea of selection selectivity control is to keep the selectivity of each selection operator constant, such that the output size of each selection operator increases linearly with the scale of underlying tables. More specifically, for non-key columns whose data distributions are specified by the user, we propose to guarantee that the new OLTP modifications on these columns adhere to the user-specified distribution function. Then, we can directly leverage the distribution function to guide the parameter instantiations. However, there also exist some columns whose distributions cannot be specified and always change over time. For example, the order shipping date (L_SHIPDATE) and receiving date (L_RECEIPTDATE) in TPC-H's LINEITEM table would dynamically change over time and exhibit different data distributions compared to the original one [12]. To address this issue, we propose a sampling-based method to continuously track the distribution changes. Specifically, we propose to use the reservoir sampling [5, 69] algorithm, which provides a scalable way to randomly choose a batch of samples in only one pass over the underlying streaming data. Then, we can use the small batch of sampled data to approximate the real data distributions, which is further leveraged to help instantiate query parameters. Based on the Hoeffding's Inequality [20], the reservoir size N can be calculated by $\frac{ln2-ln(1-\hat{\rho})}{2\delta^2}$ according to the error bound δ and confidence level $\hat{\rho}$. Note, the sampled data items are naturally sorted by the ordered map to facilitate deriving the distribution of any data range. Next, we discuss how to instantiate the parameters in the selection predicate P according to the underlying data distributions. For ease of presentation, we mainly discuss the case in which P follows the conjunctive normal form (CNF), as depicted in Eq. 8. Note, any other form of predicate can be transformed to CNF [48].

$$P = \wedge_{x=1}^{n} clause_x \quad s.t. \quad clause_x = \vee_{y=1}^{m} literal_{xy} \tag{8}$$

Here, $literal_{xy}$ can be a unary predicate operating on a single non-key column or an arithmetic predicate operating on multiple non-key columns through an arithmetic function $g()$. For example, the predicate P=(A_1<r_1 ∨ A_4>l_2) ∧ (A_2+A_3<r_3) has two cluases, in which A_1<r_1 and A_4>l_2 are two unary predicates while A_2+A_3<r_3 is an arithmetic predicate. Next, we discuss how to control the selectivity of a predicate with various numbers of clauses and literals.
Case 1: n=1 and m=1. This indicates that the predicate P has only one literal. For the unary predicate, it can be further classified into two cases according to the type of comparator contained in the predicate. The first case includes the operators of <, >, ≤, ≥, and *between and*, whose predicate seeks to cover a range of data satisfying a specific selectivity. For example, considering the predicate P=($l_k < A_k \leq r_k$) in table R_i, we need to properly instantiate the two parameters l_k and r_k so that $\sigma_P(R_i)$ satisfies the selectivity α. Recall that the data distribution of A_k is already known, l_k and r_k can be easily derived by solving the equation $F_{A_k}(r_k)$-$F_{A_k}(l_k)$=α. Here, $F_{A_k}(l_k)$ is the cumulative distribution function of A_k, which denotes the probability of a row in R_i whose attribute A_k takes a value less than or equal to l_k. The second case includes the operators of =, ≠, (*not*) *in*, whose predicate seeks to filter out specific values which meet the requirement of selectivity α. For example, considering the predicate P=(A_k=v_k), the parameter v_k can be derived by finding a value v_k in A_k satisfying $f_{A_k}(v_k)$=α. Here, $f_{A_k}(v_k)$ is defined as the probability of a row in R_i whose attribute A_k takes a value v_k. For the arithmetic predicate, it can be generalized as P=$g(A_{i_1}, \cdots, A_{i_v})$, where $g()$ is an arithmetic function which operates on multiple non-key columns $A_{i_1}, \cdots, A_{i_v}$ in table R_i. For this case, we propose to use the arithmetic function $g()$ to precompute and dynamically update the result data distribution based on the data distributions of involved non-key columns. Then, we can instantiate the parameter according to the result data distribution. For example, considering the predicate P=A_2+$A_3 \leq r_3$ that operates on two non-key columns A_2 and A_3 in R_i. We can compute the result distribution by adding the sampled column values of A_2 and A_3. Then, to meet the requirment of selectivity α, the parameter r_3 can be derived by finding a value v_r satisfying $F_{A_2+A_3}(v_r)$=α, where $F_{A_2+A_3}$ denotes the cumulative distribution function of $A_2 + A_3$.
Case 2: n>1 and m=1. This indicates that the predicate P has n clauses, each of which has only one literal. Suppose the predicate P involves non-key columns $A_{i_1}, \ldots, A_{i_v}$ in table R_i. We first group these columns according to their correlations. Following previous studies [19], the columns are independent of each other by default, unless the user explicitly specifies column dependencies. Note, as the result data distribution of an arithmetic predicate is determined by the columns involved in the arithmetic function $g()$, then $g()$ would also introduce correlations between these columns. Thus, we should put both the columns involved in $g()$ and the result of $g()$ into the same group. Then, we maintain the joint data distribution for each group if it contains more than one column. Otherwise, we maintain only the individual data distribution for each group's column. Finally, based on the dependency between each group and the groups' data distributions, we can find valid parameter values meeting the requirement of selectivity α. For example, considering the predicate P=(A_1<r_1) ∧ (A_1+A_2<r_2) ∧ A_3<r_3, suppose A_1, A_2 and A_3 are independent of each other. However, as the result of A_1+A_2 depends on both A_1 and A_2, we put them into the same group and maintain the joint data distribution of A_1, A_1+A_2 and A_2. Note, we put A_2 in the group to facilitate updating the join data distribution under dynamic modifications. In addition, A_3 is put in another group and its data distribution is maintained individually.
Case 3: n=1 and m>1. This indicates that the predicate has one clause that contains more than one literal. According to De Morgan's Law [13], it can be converted to case 2 by performing a negation on the predicate P. For example, considering a predicate P= $\vee_{y=1}^{m}$ $literal_y$ with selectivity α, we can convert it into $\overline{P}$= $\wedge_{y=1}^{m}$ $\overline{literal_y}$ with selectivity 1-α.
Case 4: n>1 and m>1. This indicates that the predicate has several clauses and some clauses have more than one literal. Since our goal is to guarantee the selectivity of the whole predicate P, rather than guaranteeing the selectivity of each sub-predicate or subsub-predicate, we propose to leverage the following two rules in the area of set theory to help simplify P.

$$rule_1 : \sigma_{literal_i} \cup \sigma_{literal_j} = \sigma_{literal_j} \quad if \quad \sigma_{literal_i} \leftarrow \emptyset$$
$$rule_2 : \sigma_{clause_i} \cap \sigma_{clause_j} = \sigma_{clause_j} \quad if \quad \sigma_{clause_i} \leftarrow U$$

From $rule_1$ and $rule_2$, we can eliminate any *clause* and *literal* without affecting the selectivity of P if their selection results meet specific conditions, such as the empty set ∅ and universal set U. To this end, we propose to assign boundary values to the parameters in *clauses* and *literals* according to the comparators involved (see Table 1). Then, based on the two rules, we can successfully reduce the complex predicates into three simple cases (i.e., cases 1, 2, and 3) described above. More specifically, we first try to set each σ_{clause_i} as the universal set U. If all σ_{clause_i} can be set as U, we would retain only one clause and eliminate other clauses, which successfully reduces our case to case 3 (the retained clause has more than one literal) or case 1 (the retained clause has exactly one literal). Otherwise, we retain all clauses that could not be eliminated. Next, for each $clause_x$ of the remaining clauses, we try to set each $\sigma_{literal_{xy}}$ constructed by its literals as the empty set ∅. Then, similar to the clause elimination procedure, we retain one literal in each clause. In this way, we can reduce our case to case 2. Taking the predicate P=(A_1<r_1 ∨ A_4>r_2) ∧ (A_2+A_3<r_3) with two clauses as an example, we can set $\sigma_{A_1<r_1 \vee A_4>r_2}$ as the univerasl set U by assigning r_1=+∞. Then, we can eliminate the first clause and reduce it to A_2+A_3<r_3. That is, we only need to instantiate the parameter r_3 satisfying $Sel(\sigma_{A_2+A_3<r_3})$=$\alpha$, where α is the required selectivity.
Complexity Analysis. Finally, we analyze the time and space complexity of *Vodka* in tackling adaptive parameter instantiation. Suppose there are $K + A + D$ literals in all query templates, where K literals are unary predicates over single non-key columns with dynamic distribution changes, A literals are arithmetic predicates over multiple non-key columns, and D literals are unary predicates over multiple dependent non-key columns, with an average group

size of m dependent non-key columns per group. Let N denote the size of the reservoir, and $s_{reservoir}$ denote the space cost in each reservoir for a single non-key column or arithmetic value. The overall storage complexity for maintaining these literals is calculated as $O(s_{reservoir} \cdot (K + A + m \cdot D))$. In practice, the reservoir is implemented as an order statistic tree, which is a variant of the binary search tree that supports finding the $i-th$ rank of an element with $O(logN)$ in addition to effective insertion, lookup, and deletion [68]. For the time complexity, the initial setup of the reservoirs requires $O(NlogN)$ for sorting initial data in order. The process of simplifying predicates needs to assign boundary values to the parameters in literals, thus its time complexity is $O(K + A + D)$. Subsequent updates to the reservoir adhere to $O(logN)$ complexity per insertion and maintain orderliness. When instantiating parameters for each query, the time complexity is $O(logN)$ for determining the target position of the reservoir and returning the corresponding parameter for each literal.

4.3 Join Cardinality Control

Having guaranteed the selectivity of each selection operator, we now discuss how to ensure that the join cardinality size changes linearly with the table size under dynamic modifications. Since the *PK-FK* joins take the largest proportion of join operators in industry-grade benchmarks, we focus on the *PK-FK* join in this paper and leave the *FK-FK* join as our future work. For example, all 33 and 197 join operators contained in SSB and JOB-LIGHT benchmarks are *PK-FK* joins [19, 41, 66]. Among the 65 join operators in TPC-H benchmark, only 2 operators are *FK-FK* joins [56, 59]. Moreover, even if a query in TPC-H contains the *FK-FK* join operator, we observe that it only appears at the upper level of the query tree [59] with small cardinality sizes. This indicates that the *FK-FK* join operator processes a small amount of data and has negligible impacts on the query performance. In the following, we assume that R_i is the referenced table containing primary keys, and R_j is the referencing table containing foreign keys. More specifically, the output size of different join types depends on two kinds of cardinalities, which are join cardinality and join distinct cardinality. A join cardinality n_{jc} requires that there exactly exist n_{jc} matched pair of rows in the join input, while a join distinct cardinality requires that there exactly exist n_{jdc} primary/foreign key values in n_{jc} matched pairs. For example, the output size of an equal-join and a semi-join can be represented as n_{jc} and n_{jdc}, respectively. Moreover, the output size of other join types can be represented as a linear combination of n_{jc} and n_{jdc} [65]. Therefore, in this section, we discuss the cardinality change of n_{jc} and n_{jdc} using equal-join (denoted as $\bowtie_=$) and semi-join (denoted as $\ltimes$) as examples. For ease of presentation, we use $\bowtie_J$ to represent both of the two join operators unless differentiation is necessary. Then, our aim is to manipulate the modifications such that the change of join cardinality size follows the form:

$$|R'_i \bowtie_J R'_j| = (c_1 \cdot \frac{|R'_i|}{|R_i|} + c_2 \cdot \frac{|R'_j|}{|R_j|} + c_3) \cdot |R_i \bowtie_J R_j|. \tag{9}$$

Here, R'_i and R'_j are the new tables after modifications, c_1, c_2, and c_3 are the constant coefficients. Next, based on Eq. 10, we further classify the join scenario into four cases according to whether there exists delta data in each joined table.

$$\begin{aligned} &|R'_i \bowtie_J R'_j| = |(R_i \cup \Delta R_i) \bowtie_J (R_j \cup \Delta R_j)| \\ &=|(R_i \bowtie_J R_j) \cup (\Delta R_i \bowtie_J R_j) \cup (R_i \bowtie_J \Delta R_j) \cup (\Delta R_i \bowtie_J \Delta R_j)| \end{aligned} \tag{10}$$

Case 1: ΔR_i=$\emptyset$ **and** ΔR_j=$\emptyset$**.** This indicates that both of the joined tables do not involve delta data. Thus, the size of join cardinality n_{jc} and join distinct cardinality n_{jdc} would keep unchanged, i.e., $|R_i \bowtie_J R_j|$=$|R'_i \bowtie_J R'_j|$. That is, c_1=0, c_2=0, and c_3=1 in Eq. 9.
Case 2: $\Delta R_i \neq \emptyset$ **and** ΔR_j=$\emptyset$**.** This indicates that only the referenced table R_i has delta data in the joining process. Note, when a delete operation occurs in R_i, R_j would also need to delete foreign keys that reference the deleted *PK*s in R_i due to the referential integrity constraint [38]. Since R_j does not have delta data in this case, we can derive that ΔR_i only contains newly inserted *PK*s. Moreover, as the new *PK*s could not join with any *FK* in R_j, we can deduce that $|(R_i \cup \Delta R_i) \bowtie_J R_j|$=$|(R_i \bowtie_J R_j) \cup (\Delta R_i \bowtie_J R_j)|$=$|(R_i \bowtie_J R_j) \cup \emptyset|$=$|R_i \bowtie_J R_j|$. That is, c_1=0, c_2=0, and c_3=1 in Eq. 9.
Case 3: ΔR_i=$\emptyset$ **and** $\Delta R_j \neq \emptyset$**.** This indicates that only the referencing table R_j has delta data that participate in the joining process. Then, the join result is deduced as $|R_i \bowtie_J (R_j \cup \Delta R_j)|$=$|(R_i \bowtie_J R_j) \cup (R_i \bowtie_J \Delta R_j)|$. For equal-join, since the *FK*s in R_j all come from the *PK*s in R_i, then we have $|R_i \bowtie_= R_j|$=$|R_j|$ and $|R_i \bowtie_= \Delta R_j|$=$|\Delta R_j|$. Thus, the cardinality size of equal-join is $|R'_j|$. That is, c_1=0, c_2=1 and c_3=0 in Eq. 9. For semi-join, as the *FK* values usually follow a user-specified distribution when *PK*s are fixed [19, 59], we propose to keep modifications of *FK* values comply with the specific distribution. With this method, we can infer that the join result $R_i \ltimes \Delta R_j$ must be a subset of $R_i \ltimes R_j$. Consequently, the output size of a semi-join is still $|R_i \ltimes R_j|$. That is, c_1=0, c_2=0, and c_3=1 in Eq. 9.
Case 4: $\Delta R_i \neq \emptyset$ **and** $\Delta R_j \neq \emptyset$**.** This indicates that both the referenced and referencing tables have delta data that participate in the joining process. In this complex case, to ensure the non-decreasing cardinality size of each join operator when scaling the data size, previous stuides [11, 12, 39, 75] do not perform any delete operation on join tables involved in this complex case. This is because delete operations on a referenced table usually lead to cascading deletes due to the referential integrity constraint, which might further lead to a sharp decrease in the join cardinality size. Therefore, we propose to follow these studies and use the *distributive law* [3] to deduce the join result as follows.

$$\begin{aligned} &|(R_i \cup \Delta R_i) \bowtie_J (R_j \cup \Delta R_j)| \\ &= |(R_i \bowtie_J R_j) \cup (\Delta R_i \bowtie_J R_j) \cup (R_i \bowtie_J \Delta R_j) \cup (\Delta R_i \bowtie_J \Delta R_j)| \\ &= |(R_i \bowtie_J R_j) \cup \emptyset \cup (R_i \bowtie_J \Delta R_j) \cup (\Delta R_i \bowtie_J \Delta R_j)| \\ &= |(R_i \bowtie_J R_j) \cup ((R_i \cup \Delta R_i) \bowtie_J \Delta R_j)| \end{aligned} \tag{11}$$

From Eq. 11, we can see that the join result is closely related to the delta data in both tables. As there exist new *PK* values inserted into the referenced table and these *PK* values would also participate

Table 1: Boundary Values in Selection Predicates

Operator		>, ≥	<, ≤	in, =	not in, ≠
Query Parameters	U	$-\infty$	$+\infty$	/	NULL
	$\emptyset$	$+\infty$	$-\infty$	NULL	/

in the join process, we propose first to divide the delta data in R_j into $\Delta R_j{}^{old}$ and $\Delta R_j{}^{new}$, where $\Delta R_j{}^{old}$ references the original *FK*s in R_i and $\Delta R_j{}^{new}$ references the new *FK*s in R_i. Then, we can further transform $(R_i \cup \Delta R_i) \bowtie_J \Delta R_j$ into Eq. 12. For equal-join, based on whether the right table's *FK* values exist in the left table's *PK* values, we can derive: $|R_i \bowtie_= R_j|$=$|R_j|$, $|R_i \bowtie_= \Delta R_j{}^{old}|$=$|\Delta R_j{}^{old}|$, and $|\Delta R_i \bowtie_= \Delta R_j{}^{new}|$=$|\Delta R_j{}^{new}|$. Taken together, we have the cardinality size as $|R_j| + |\Delta R_j|$. That is, c_1=0, c_2=1, and c_3=0 in Eq. 9.

$$\begin{aligned} &|(R_i \cup \Delta R_i) \bowtie_J \Delta R_j| = |(R_i \cup \Delta R_i) \bowtie_J (\Delta R_j{}^{old} \cup \Delta R_j{}^{new})| \\ &= |(R_i \bowtie_J \Delta R_j{}^{old}) \cup \emptyset \cup \emptyset \cup (\Delta R_i \bowtie_J \Delta R_j{}^{new})| \\ &= |(R_i \bowtie_J \Delta R_j{}^{old}) \cup (\Delta R_i \bowtie_J \Delta R_j{}^{new})| \qquad (12) \end{aligned}$$

For semi-join, as $\Delta R_j{}^{old}$ is only related to R_i and the distribution of $\Delta R_j{}^{old}$'s *FK* values can be manipulated to comply with the distribution of R_j's *FK* values (similar to case 3). Then, we can infer that $R_i \ltimes \Delta R_j{}^{old}$ is the subset of $R_i \ltimes R_j$, which would not change the semi-join's output size. However, as the *FK*s in $\Delta R_j{}^{new}$ references some *PK*s in ΔR_i, then $\Delta R_i \ltimes \Delta R_j{}^{new}$ must increase the semi-join's output size. Next, we discuss how to manipulate the modifications in ΔR_i and ΔR_j such that the join cardinality size changes linearly with the data size. Since the output size of a semi-join is determined by the number of distinct *FK* values, we propose to use a consistent proportion (denoted as θ_1) of new *PK*s in ΔR_i to populate the *FK*s in ΔR_j. Specifically, as $\frac{|R_i \ltimes R_j|}{|R_i|}$ represents the proportion of *PK*s in R_i that are used to populate *FK*s in R_j, we propose to set θ_1=$\frac{|R_i \ltimes R_j|}{|R_i|}$, which indicates that the proportion keeps unchanged even after performing modifications on both of the two joined tables. Note, this also implicitly indicates that we should manipulate the delta data in R_j such that $|\Delta R_j| \geq \theta_1 \cdot |\Delta R_i|$. Then, considering the output size of $R_i \ltimes R_j$ can be represented as $\theta_1 \cdot |R_i|$, then we can finally deduce the output size of $R'_i \ltimes R'_j$ as $\theta_1 \cdot |R'_i|$, which can be further represented as $\frac{|R'_i|}{|R_i|} \cdot |R_i \ltimes R_j|$. That is, c_1=1, c_2=0, and c_3=0 in Eq. 9.

4.4 Projection and Aggregation Control

The projection operator Π selects specific columns from a given input and hides all the other columns. If there exist no duplicate rows in the selected columns, then the output size equals the input size. That is, the projection operator has no effect on the cardinality size. Otherwise, it is similar to the case of 'GROUP BY' in the aggregation operator. For the aggregation operator $\mathcal{G}$, it can be broadly classified into two types. The first type uses an aggregation function like 'SUM' to perform a calculation on a set of input values and returns a single value. As this type of operator is generally the last operator in the query tree and there exists no other operator taking the last operator's output as its input, then its output size does not affect the query performance. The other type of operator like 'GROUP BY' divides the input into groups, and the aggregation function is used to return a single value for each group. That is, the output cardinality is determined by the number of distinct values in the input. On the one hand, the aggregation on non-key columns usually involves categorical data [14, 36, 65] with a small number of distinct values, then the operator's output size is also relatively small. Although there might exist other operators operating on the small-sized output, they have a marginal impact on the query performance [40]. On the other hand, the aggregation on key columns generally happens on *FK* columns. This is because each *PK* value uniquely identifies a row and the *PK* column is not of interest to the aggregation operator. However, the number of distinct values in an *FK* column is closely related to the number of *PK*s it references, and there might exist lots of distinct *FK* values that have a great impact on the query performance. Thus, we should also guarantee that its output scales linearly with the underlying table size. As the aggregation operator typically occurs after the join and selection operator, we can use a semi-join to evaluate the operator's output size since the output size of a semi-join is determined by the number of distinct values in an *FK* column. Specifically, suppose the aggregation operator's input $\mathcal{G}_{input}$ contains the *FK* column of R_j, then the output size can be calculated as $|R_i \ltimes \mathcal{G}_{input}|$. It is similar to the result of $(R_i \cup \Delta \mathrm{R_i}) \ltimes (R_j \cup \Delta \mathrm{R_j})$ of the case 4 in § 4.3, we thus can guarantee that its output size increases linearly with the scale of underlying tables.

4.5 Performance Measurement

As previously discussed, the OLTP business usually plays a mission-critical role in real-world applications, and users tend to have strict requirements of stable OLTP throughput according to the specific business scenario [37, 42, 55]. Users are primarily concerned about whether the HTAP system could mitigate resource contentions well, which ensures that the system could achieve optimal OLAP performance while meeting the requirement of a specific OLTP throughput [37, 42, 55]. Since consistent computational complexity leads to the same resource consumption pressure, to ensure a fair comparison among different HTAP systems, we propose to follow the paradigm of comparing the performance of *A*-engine under the same OLTP throughput [11, 50]. Note that although under the the same OLTP throughput leads to the same data size growth rate in the *A*-engine, the start time of each OLAP query still might differ since each *A*-engine has quite different OLAP processing capabilities. Note, the query complexity of each *A*-engine would be inconsistent since different start times lead to different table sizes and cardinality sizes in each query operator, both of which have a critical impact on the query complexity. Fortunately, given that we have guaranteed that the cardinality size (i.e., output size) of each query operator increases linearly with the scale of data size (i.e., table size), this provides us an opportunity to build a mathematical regression model to approximate the relationship between the data size and query performance for each *A*-engine. With this mechanism, we can effectively derive and compare the query performance of different systems under any data size. More specifically, as the *A*-engine has generally employed a number of optimization techniques to handle large-scale queries, the time complexity of each query operator is either $O(M)$ or $O(logM)$, where M is the input size if the operator consists of a single input; similarly, if the query operator contains two inputs with the sizes of M and N, then the operator's time complexity generally approximates $O(M+N)$ or $O(MlogM + NlogN)$ [52]. Consider that each query can be represented as a query tree, then the output of one operator serves as the input for the subsequent operator [34]. Since we have guaranteed that the output size of each query operator increases linearly with the table size, these observations motivate us to build a log-linear regression model $L(q, S) = \sum_{i=1}^{n}[a_i \cdot S_i log(S_i) + b_i \cdot S_i + c_i \cdot log(S_i)] + d$ to fit the query latency with regard to the table size. Here, $L(q, S)$ is the derived latency for query q under the data size S, n is the total number of tables, S_i is the size of i-th table, a_i, b_i, c_i and d are fitting parameters. Moreover, we use *Huber Loss* function [8, 21, 29] to reduce the impact of outliers from performance fluctuation, which makes the loss function more robust in the presence of noisy data. In this way, we can effectively compare the query performance of different systems under the same data size and query complexity.

5 BENCHMARK CONSISTENCY MODEL

5.1 Consistency Model Definition

We first introduce the basic concepts of the evolution of the data versions. On this basis, we describe three consistency models, which demonstrate the difference between the latest data version written in the T-engine and the data version being read in the A-engine.

Version Write. The write operation $w_i(x, val_i)$ denotes a committed transaction writes a data item x with the value val_i. The commit time of $w_i(x, val_i)$ is marked as t_i.

Example 1. In Fig. 5, Trx_1 and Trx_3 in TP Client$_1$ perform two writes on the data item x with $w_1(x, 0)$ and $w_5(x, 1)$, which update x to 0 and 1, respectively. The commit times of the two writes are t_1 and t_5.

Version Write Sequence. The version write sequence $S_t(x) = \{w_1(x, val_1), \cdots, w_k(x, val_k)\}$ covers all sequential version writes on data item x with commit times not larger than time t. That is, $\forall i<j$, we have $t_i<t_j \leq t$, and $\forall w_n, w_{n+1}$, we have $\nexists w_l$ $s.t.$ $t_n<t_l<t_{n+1}$.

Example 2. In Fig. 5, the version write sequence of the data item x with commit time not larger than time t_8 is $S_{t_8}(x) = \{w_1(x, 0), w_3(x, 2), w_5(x, 1), w_7(x, 3)\}$. So the version evolution of item x is 0, 2, 1, 3.

Data Version. The data version of a data item x at time t is the result of executing a version write sequence $S_t(x)$. The corresponding version stamp is defined by the length of $S_t(x)$.

Example 3. In Fig. 5, the data version of x at t_6 is 1 since the result of executing $S_{t_6}(x)$ is 1. Its corresponding version stamp is $|S_{t_6}(x)|$=3.

Next, we discuss how to define different consistent models using the data versions. The consistency models are defined according to two kinds of data version gaps: the data version gap between the T- and A-engines (denoted as CM_{TA}), and the data version gap between different sessions (i.e., clients) in the A-engine (denoted as CM_{AA}). Specifically, consider a data item x, we assume that: 1) the last write of x on the T-engine is $w(x, val_l)$ and its commit time is t_l; 2) there exist two sessions s_1 and s_2 on the A-engine, which perform two reads $r_{s_1}(x)$ and $r_{s_2}(x)$ at the same time t_r s.t. $t_r \geq t_l$; 3) $r_{s_1}(x)$ and $r_{s_2}(x)$ read the data versions generated by the write sequences $S_{t_{s_1}}(x)$ and $S_{t_{s_2}}(x)$, respectively. Here, $t_{s_1} \leq t_l$ and $t_{s_2} \leq t_l$. Then, CM_{TA} and CM_{AA} can be defined as:

$$CM_{TA}(w(x, val_l), r_{s_1}(x)) = |S_{t_l}(x)| - |S_{t_{s_1}}(x)| = g_1, \quad (13)$$

$$CM_{TA}(w(x, val_l), r_{s_2}(x)) = |S_{t_l}(x)| - |S_{t_{s_2}}(x)| = g_2, \quad (14)$$

$$CM_{AA}(r_{s_1}(x), r_{s_2}(x)) = |S_{t_{s_1}}(x)| - |S_{t_{s_2}}(x)| = g_3. \quad (15)$$

Linear Consistency refers to the data version accessed on the A-engine exactly equals to the latest data version in the T-engine. It ensures that for any A-engine's read, its read data version must be produced by all committed writes in the T-engine before the read is executed. In this consistency model, we require g_1=g_2=g_3=0.

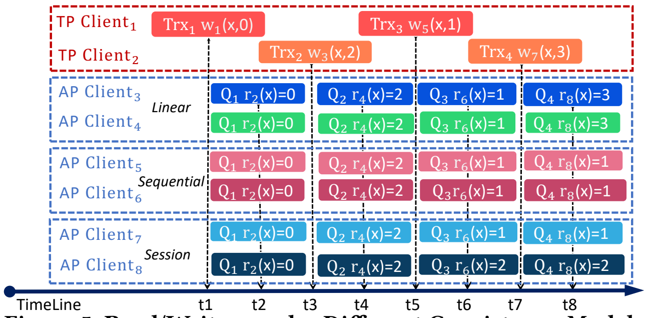


Figure 5: Read/Write x under Different Consistency Models

Sequential Consistency ensures that any two reads in different sessions of the A-engine must read the same data version if they are executed at the same time. Note, the two reads are not required to access the latest data version in the T-engine. In this case, we require g_1=g_2 $\geq$ 0 and g_3=0. For example, the $Client_5$ and $Client_6$ in Fig. 5 perform two reads on x at t_8, and both of them read the data version with x=1, rather than the latest version of x=3.

Session Consistency allows each session of the A-engine to read different data versions even if the reads are executed at the same time. In this case, we only require $g_1 \geq 0$ and $g_2 \geq 0$.

As described above, the three consistency models concern the version gap of accessed data between T- and A-engines in essence. Therefore, the measurement of freshness should precisely reflect the degree of this version gap. In light of this, for any given data item, we describe its freshness as the gap between its latest commit time in the T-engine and the commit time of its visible version in the A-engine. Further, the freshness of a query Q is defined as the maximum gap for all the query's accessed data items. That is Eq. 16, where R_Q denotes the set of data items returned by the query Q, $ts_T(d_i)$ is the latest commit time of data item d_i in the T-engine, and $ts_A(d_i)$ is the commit time of d_i's visible version in the A-engine.

$$freshness(Q) = max\{ts_T(d_i) - ts_A(d_i) | \forall d_i \in R_Q\} \quad (16)$$

5.2 Measurement of Freshness

To measure the freshness of a query Q, one naive approach is to first execute Q on the T- and A-engines simultaneously, and then use the query results on the two engines and Eq. 16 to compute Q's freshness. However, this approach incurs additional query execution costs in the T-engine. Moreover, it also poses a challenge in obtaining each result data item's latest commit time in the T-engine and the commit time of its visible version in the A-engine. The ideal method to obtain these timestamps is referring to the MVCC version chain inside the database. However, this internal state is usually invisible to the users. Another approach of scanning transaction logs to obtain the timestamp of each data version can cause considerable IO costs and degrade the system performance [18]. Therefore, to minimize the impact on the system performance, a non-intrusive method is to log the whole version evolutions for each data item on the client side. However, it is still difficult to identify the version of each data item in the query result. This is because a value might exist multiple times in the version evolution of a data item, while the query result only contains data values. For example, when the query in the A-engine returns a data item x with the value of 1 and the version evolution of x in the T-engine is $1 \longrightarrow 2 \longrightarrow 1 \longrightarrow 3 \longrightarrow 1$, then it's ambiguous which version is touched by the query. Thus, we need to attach the version information for all data items in the query result. To this end, the intuitive approach is to attach a version column for each attribute in both of T- and A-engines. Note, we track the version evolution of each column value instead of each tuple. This is because users are generally concerned about whether they could access the latest data version within the expected query columns [6, 46], while the version evolution of a tuple does not indicate that all of its attribute values are actually evolved (i.e., changed). However, attaching version columns would also incur high update and storage costs on the system. To address this issue, we propose a decoupled approach that only attaches one version column in each table to track the tuple-level version evolution on the server side, and leaves the expensive track of column value's version evolution on the client side. In this way, we can effectively reduce the performance interference in the two engines. Specifically, on the server side, the version column in each table helps identify the version of data touched by the query at the granularity of tuples. On the client side, as shown in Fig. 6, it maintains the actual evolution of each column value along with the evolution of its corresponding tuple. Note, the commit timestamp of each tuple-level data version is also maintained on the client side.

Example 4. In Fig. 6, T_1 is stored in two engines and is attached to a tuple-level version column. Suppose that query Q on the A-engine accesses the attribute c_1 with "Select c_1 from T_1 where PK=1 or PK=4", which indicates that it accesses two c_1 values in two tuples with PK=1 or 4. Although the versions of two tuples in the A-engine lag behind the latest versions, the versions of column values touched by the query are consistent in the two engines. This is because the version evolution of the two tuples is caused by the updates in the column c_2. Thus, we can derive that the query Q returns the freshest result, i.e., $freshness(Q)$=0. However, if the query Q accesses the attribute c_2 with "Select c_2 from T_1 where PK=1 or PK=4", we can see that the version of column values touched by the query lags behind the version in the T-engine. For ease of presentation, we use ts_i^j to denote the commit time of the i-th version for the tuple with PK=j. Then, for the c_2 value in the tuple with PK=1 (resp. PK=4), the commit time gap between the A-engine version and the T-engine version is $ts_4^1 - ts_2^1$ (resp. $ts_3^4 - ts_2^4$). Finally, according to Eq. 16, $freshness(Q)$ is calculated as $max(ts_4^1 - ts_2^1, ts_3^4 - ts_2^4)$.

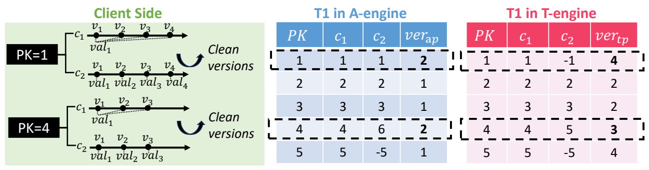
Note that the client side usually maintains the version evolution for a small amount of recently updated column values. There are three primary reasons. First, we only track the version evolution for columns both written by transactions in the T-engine and then accessed by queries in the A-engine. Second, we only track the column values that have been modified, and static data does not require version tracking. Third, for any column value maintained on the client side, once a specific version has been touched by the query in the A-engine, we can safely prune all versions older than that version. This is because we can infer that the accessed version has been successfully synchronized to the A-engine, and all the older versions would not help the measurement of freshness.

Complexity Analysis. We now analyze the time and space complexity for freshness calculation. The time complexity is $O(N)$, as it iterates to verify version status for returned results, where N is the size of accessed rows. For space complexity, on the server side, it is required to record the global version evolution by attaching column ver with the storage cost as s_1, and then its storage complexity is only related to the size of table T, i.e., $O(s_1 \cdot |T|)$, which is not prominent compared to the whole table.

On the client side, the storage cost will not experience uncontrolled growth. Above all, we only maintain and track version evolution for columns written by transactions and then accessed by queries simultaneously. Suppose the number of such columns is k_c in table T, and the storage of one column's version information per row is s_2 so that the storage cost is thus $s_2 \cdot k_c$ for each row. Second, we only maintain the rows that will undergo modifications, so that static data needn't have version management. Suppose the number of rows inserted and modified per second is N_{ins} and N_{mod}. Then the number of new versions generated per second is $(N_{ins} + N_{mod}) \cdot k_c$. Third, let Pos_q be the version position accessed by query q, and N_q be the number of rows accessed per query per second. Pos_q tells the version having been synchronized from T-engine to A-engine, which means the versions before Pos_q are not available by queries and can be safely deleted from the client side. Therefore, the version deletion rate is $k_c \cdot (Pos_q - 1) \cdot N_q$. So, in each time unit, the overall storage cost, in theory, is $O(s_2 \cdot k_c \cdot (N_{ins} + N_{mod} - (Pos_q - 1) \cdot N_q))$. Note that in practice, k_c, N_{ins}, N_{mod}, N_q and Pos_q can be all obtained from benchmark definition. Meanwhile, in practical HTAP systems, the version lag between two engines is not significant, and versions prior to the one read can be quickly cleared so that the client side approximates maintaining version evolution from a recent short period. Therefore, the storage cost remains controllable and stable.

6 BENCHMARK DATA SHARING

Data sharing plays a critical role in ensuring that the data produced can be transferred to the A-engine in time for real-time analytics. To improve the customer experience, all HTAP systems are dedicated to enhancing the data sharing efficiency [55]. The users are usually concerned with the real-time synchronization status of the system when performing their analytic tasks. For example, many applications (e.g., stock price monitoring) require real-time analytics on fully synchronized data before a specific time point. Therefore, it is essential to realize whether the data written before a certain time point have been fully synchronized to the A-engine [9, 51, 64]. This inspires us to design a benchmark method that measures the synchronization latency of the system at different time points.



T1 in A-engine

PK	c_1	c_2	ver_{ap}
1	1	1	**2**
2	2	2	1
3	3	3	1
4	4	6	**2**
5	5	-5	1

T1 in T-engine

PK	c_1	c_2	ver_{tp}
1	1	-1	**4**
2	2	2	2
3	3	3	2
4	4	5	**3**
5	5	-5	4

Figure 6: Example of Freshness Measurement

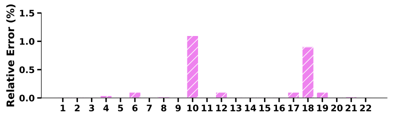


Figure 7: Relative Errors of Cardinality

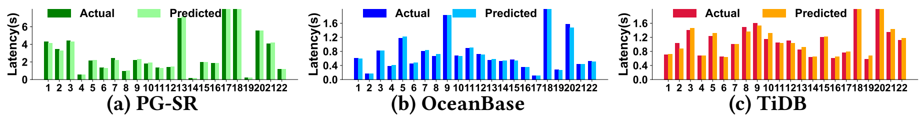


Figure 8: Actual & Predicted Latencies For All OLAP Queries

Specifically, we first propose to randomly select a number of test time points T_i. Then, for each time point T_i, we define its synchronization latency $sync(T_i)$ as the time elapsed from T_i when all the T-engine's updates committed before T_i are visible to the A-engine. Finally, the data sharing capacity of the systems is measured as the average synchronization latency of all the n test time points, i.e., $\frac{1}{n}\sum_{i=1}^{n} sync(T_i)$. Moreover, the speed of fresh data generation is mainly determined by the throughput and the write ratio in the T-engine. We thus regulate each T-engine to exhibit the same throughput and workload semantics such that each system experiences the same speed of fresh data generation and the same synchronization pressure. Meanwhile, the synchronization pressure can be tuned on demand by changing the throughput and write ratio in the workload [46]. However, since the internal synchronization status is invisible to the users, it imposes a great challenge of exactly measuring the synchronization latency. In addition, the method of instrumenting system kernels to catch the internal execution and synchronization state is laborious or even impossible, especially for closed-source commercial systems [53]. As the HTAP systems generally ensure the linearizability of writes [4, 31, 71], this indicates that the order of synchronization (data visibility) is consistent with the commit order of writes in the T-engine. In light of this, we only need to track the synchronization state of the last committed write before the test time point. That is, once the modifications from the specific last committed write become visible in the A-engine, we can guarantee that all prior writes before the test time point have been synchronized. To this end, we need to address two critical issues. First, we need to catch the timestamp of the T-engine's last committed write before each test time point, and then determine the overall synchronization status based on this write. Second, we need to exactly track the version evolution of this write in the two engines, such that we can identify whether the A-engine has successfully synchronized this write. For the first issue, we propose to monitor the commit timestamps of writes on the client side and also maintain a variable that identifies the write containing the largest witnessed timestamp that is smaller than the test time point. Note, the information of monitored writes before the test time point can be promptly cleared once the time point passes. For the second issue, similar to the method used in measuring freshness, we propose to attach a version column to each table to indicate the version evolution of tuples in the two engines. Specifically, each write changes the version evolution of the tuple it modifies. If the A-engine's tuple version touched by a query is consistent with the tuple version of the specific last committed write, we can infer that all the writes before the test time point are successfully synchronized. Based on this observation, we propose to launch a lightweight point query at the test point time. The query retrieves the last committed write to verify whether its committed version has been synchronized to the A-engine. If not, the A-engine will retry the query process until the committed write is visible to the query. Note that this lightweight verification facilitates rapid responses with quite low query latency, thereby ensuring that the actual synchronization completion time can be caught just in time.

7 IMPLEMENTATION

The technical designs introduced before can be generally applied to compose new benchmarking scenarios. Since TPC-C or TPC-H have been popularly used to depict an HTAP scenario [11, 12, 23, 39], we also construct the HTAP scenario from these two benchmarks.
Data and Schema. We follow *CH-benchmark* [12] by replacing the relations *PartSupp*, *Part*, *Orders* and *LineItem* in TPC-H with *Stock*, *Item*, *Order* and *OrderLine* in TPC-C, which have almost the same application semantics. Tables with the same semantics share the schema by unifying the attributes from both TPC-C and TPC-H. Since the performance of A-engine is highly related to data distribution, we then make the data distribution of TPC-C follow TPC-H. Lastly, to check data consistency between two engines, tables that are subject to modification are augmented with a column ver to track data version changes.
Hybrid Workloads. We have incorporated the original 5 transactions in TPC-C and all 22 queries in TPC-H. We construct a real-time query named *ConsistencyCheck* as 'SELECT * FROM OrderLine WHERE OL_RECEIPTDATE BETWEEN ? AND ?' to check out systems' freshness on the *OrderLine* table, which undergoes the most frequent updates. Note that parameters for OL_RECEIPTDATE are used to control access to different sizes of modifications. Moreover, in the original TPC-H benchmark, an attribute named $ol\_receiptdate$ will not be modified (NULL in new tuples) by any transaction, making that newly inserted tuples are never returned/accessed in queries. This fails to meet the requirements for reading new modifications in HTAP scenarios [23]. Therefore, all previous work choose to remove many original attributes from the TPC-H schema and queries, thereby altering its elaborate query semantics. The static dimension tables, e.g., $Country$, $Region$, and $District$, have no data change. However, fact tables like $OrderLine$ accessed by A-engines are expected to read new modifications from the T-engine. Otherwise, reading a fixed data range without considering new changes would be meaningless [23]. To tame this issue, we introduce a new transaction called *ReceiveGoods*, to complete the order lifecycle. Specifically, in TPC-C, the New-$Order$ transaction creates a new order, and the $Delivery$ transaction ships this order to the customer. We thus add a new transaction, *ReceiveGoods*, to simulate the customer's actions, including confirming the date of received orders ($ol\_receiptdate$), rejecting goods or not (ol_return_flag), and providing feedback ($o\_comment$).

8 EXPERIMENTAL EVALUATION

We run extensive experiments to illustrate the effectiveness of *Vodka* and explore the capabilities of three kinds of open-source HTAP systems, which are PostgreSQL Streaming Replication (PG-SR v15.0) [44], OceanBase v4.2.0 [74] and TiDB v7.1.0 [22].

Cluster Configuration. Each system is deployed on three servers, with each having an Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz with 96 physical cores, 375GB RAM, and a 1.5TB pmem disk. *Vodka* is deployed independently on a client node with the same configuration. All nodes are connected via 10GbE network cards.

Benchmark Configuration. By default, all experiments are performed on a data size of 10GB, i.e., 120 warehouses with each T-thread bound to one warehouse [46, 76]. The default reservoir sampling size is 4 million ($4M$) rows according to Hoeffding's Inequality with its theoretical error bound $\delta = 0.1\%$ in a confidence level of $\hat{\rho} = 99.9\%$. By default, we let query *ConsistencyCheck* read new modifications within 5 seconds (secs). Selectivity α for each query is calculated from the initial round of queries on A-engine. To avoid code-start, all experiments have a 5-minute warm-up.

Database Configuration. PG-SR is deployed based on streaming replication mode, which assigns a single master node for OLTP tasks, a synchronous backup slave for disaster recovery, and an asynchronous slave for OLAP tasks with the synchronization interval defaultly set to 0 sec. OceanBase is deployed to serve OLTP and OLAP tasks on each node simultaneously. For TiDB, its TiKV (write) and TiFlash (read) are deployed on each server by manually bound to the CPU cores [10, 37] for OLTP and OLAP tasks respectively. We conduct an experiment to discover the maximum OLTP throughput (TPS) for three systems under 120 T-threads with one A-thread which has the minimum impact to T-engines. PG-SR, OceanBase, and TiDB achieve their highest TPS of 6.5K, 5K, and 2K, respectively. To make a fair comparison among HTAP systems, we set the TPS of T-engines no larger than 2K; to guarantee a stable TPS (=2K) for the three databases, the maximum number of A-threads is 10. Thus, by default, we set TPS=2K and A-threads=10.

Baselines. The most state-of-the-art work is *HATtrick* [39] and *Hybench* [75], both of which have similar definitions to evaluate *resource isolation* and *consistency model* while having distinct schema and workloads. Since *Vodka* has constructed an HTAP scenario from the popular TPC-C and TPC-H, for a fair comparison, we select *HATtrick* as the baseline, which constructs a similar HTAP scenario based on TPC-C and SSB (a simplified TPC-H).

8.1 Evaluation of the Key Design of *Vodka*

Cardinality Control of OLAP Queries. We adopt average relative error $Err = \frac{1}{|Q_{op}|}\frac{\sum_i ||C_i| - |\hat{C}_i||}{\sum_i |C_i|}$ [33] to measure the accuracy of cardinality control to all operators ($|Q_{op}|$) in queries Q, where $|C_i|$ and $|\hat{C}_i|$ are the runtime real and the expected cardinalities of the i-th operator under a specified selectivity α ($\in [0, 1]$). A smaller Err means a better control. Since the data change is decided by the throughput of T-engine and our cardinality control methods are insensitive to databases, we take PG-SR to illustrate the average relative error for the queries in *Vodka.* We run *Vodka* for 5, 10, 15 mins, and collect the operator cardinalities in A-engine. The average Err in Fig. 7 shows all 22 queries have low relative errors (Err is < 2%), i.e., an effective control of cardinalities.

Latency Modeling. Our log-linear regression model is designed to fit query latencies under changing data sizes. Since HTAP systems have performed differently for distinct queries, we tune the optimal parameter θ in the *Huber Loss* function for each query. We gather 100 pairs of the latency and the calculated data size every 6 secs as the training set and then collect another 50 pairs as the test set. To validate the effectiveness of the model, we compare the average actual latency with the average predicted latency in Fig. 8 on the test set, and our model performs well with the maximum performance deviation < 5% for each system.

Ablation Study. Note that, 1) we conduct reservoir sampling (F1) to track the data distribution change; 2) we have attached a column *ver* to each table on the server side and recorded the version commit timestamps (F2) on the client side to trace data version evolutions for freshness calculation; 3) we employ a lightweight point query (F3) to repeatedly check the synchronization status and measure synchronization latency. We then assess the impact of these designs on the system performance of T-engine by exposing the change of peak throughput of three systems without extra A-thread. The result is shown in Fig. 9, where features are turned on in order. We find that these new designs impose little overhead on the system performance since most information collection is asynchronously updated and maintained on the client side. The invasion of the system is relatively minimal, and the maximum side effect is < 5%.

8.2 System Evaluation

Resource Isolation. To benchmark resource isolation, the prerequisite is to ensure that workloads among systems have consistent computational complexity, i.e., resource demands. Specifically, each system under test is required to meet an identical query complexity (cardinality) under the same data size when data grows. Therefore, we compare query cardinality control of *Vodka* and *HATtrick* by their running latency changes as data grows. Given that *HATtrick* is built on a star schema, which does not follow the commonly used 3NF design paradigm of the relational schema, and the columns accessed by its OLAP workload are seldomly modified, which does not sketch a real dynamic HTAP scenario, we thus implement its approach to $Vodka$'s complex OLAP workload. We employ Q5 and Q8 of TPC-H, the representative multi-table join queries, which have a linear increase of data accessing under dynamic data changing, to demonstrate the output cardinality and the latency in Fig. 10a and Fig. 10b. Since *HATtrick*'s method restricts the query to access attributes with unchanged data/distributions and fixes the query parameters to conduct tests, it exposes an approximate constant query latency. But this stability does not benefit from resource isolation, for queried data has nothing to do with the continuous writings of T-engine. If we take a random way to assign parameters when the query starts, it produces uncertain cardinality sizes, which makes query computational complexity uncontrollable. *Vodka* achieves approximate linear changes in cardinalities along with the dynamic change of data, i.e., a well-controlled computational complexity.

We next demonstrate the OLAP performance (QphH) under different TPS by changing A-threads in Fig. 11. For a given A-thread, if increasing TPS, OceanBase experiences a relatively evident drop of per-thread QphH. This is caused by OceanBase's MVCC mechanism, which constructs longer version chains for a larger number of writings. It then lowers OLAP scan efficiency. TiDB, with its separate columnar replica and partition pruning in TiFlash, shows its performance is less sensitive to the increasing of data sizes for the stability change of query cardinality. PG-SR has the stablest per-thread QphH which is constrained by Q18 (>50s) containing a

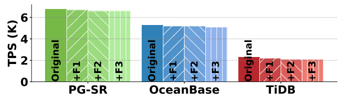
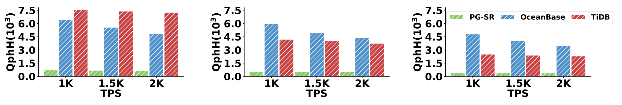


Figure 9: Ablation Study of F1, F2 and F3 in *Vodka*


(a) 1 *A*-Thread (b) 5 *A*-threads (c) 10 *A*-threads
Figure 11: Performance of *A*-engine

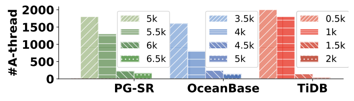
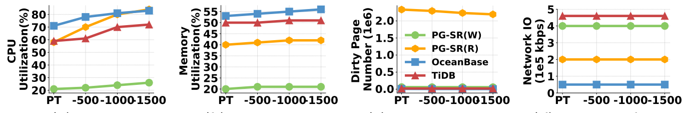


Figure 12: The Maximum *A*-thread Number


(a) CPU (b) Memory (c) Dirty Page (d) Network IO
Figure 13: The Variation of Resource Utilization

complex subquery, and PG-SR performs poorly in handling such queries. When increasing *A*-threads, all systems have per-thread QphH regressions, among which TiDB is the worst. This is primarily due to its severe multi-thread resource contention, especially CPU. PG-SR has thorough storage isolation, resulting in the least decline of QphH when increasing *A*-threads. Besides, OceanBase's slower QphH regression benefits from its optimization of *small table broadcasting* by replicating small tables across all servers, reducing distributed queries when increasing *A*-threads. We further dive into exploring the isolation ability from the view of resource contention. Specifically, since each system has a different TPS frontier (PT, Peak TPS) by the given *T*-threads, resource contentions from *A*-engine vary under different OLAP pressures. We first conduct tests to find the maximum supported/saturated *A*-threads that can keep the stable TPS for each system. In such cases, it usually meets the most intensive resource contentions between the two engines. The result is shown in Fig. 12. Next, we explore the variations in resource utilizations of the HTAP systems under their maximum supported *A*-threads as TPS changes in Fig. 13, gradually reducing PT by 500 TPS. Since PG-SR employs separate write (W) and read (R) nodes for both write and read, we show PG-SR's resource utilization for W and R separately. For all systems, the CPU is severely consumed while memory remains sufficient when the system reaches a saturated state under different TPS. Note that *A*-threads are all run on PG-SR (R) and PG-SR (W) costs the minimal CPU. But PG-SR is critically suppressed by the network usages from PG-SR (W)'s asynchronous synchronization to the two slave nodes and the large number of dirty pages in PG-SR (R), i.e., abundant historical data versions in its local cache. Besides the data synchronization between TiKV and TiFlash, TiDB requires confirming the consistency of data versions between the two engines frequently, causing severe network communication. For OceanBase, it can adaptively configure CPU for query processing, which causes its stable high CPU usages; but at its PT, the accumulation of data versions requires frequent version management, which may affect the performance of *T*-engine. Therefore, it cannot support more *A*-threads even when the CPU is not completely utilized at PT.

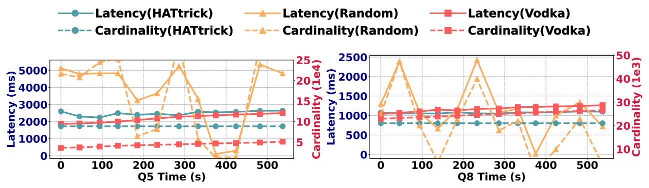
In summary, the CPU is the most critical resource under different isolation methods. Besides, physical isolation performs the best for resource isolation, but it can be bottlenecked by network IOs and cache management under high TPS. Logical isolation has provided the opportunity for adaptive resource scheduling *w.r.t* the hybrid workload, but it is bothered by retrofitting MVCC in garbage collection and version chain searches.

Consistency Model. We first compare *Vodka* and *HATtrick* across two representative scenarios to highlight their freshness benchmarking ability by using the real-time query *ConsistencyCheck*. Since OceanBase and TiDB adopt the linear consistency model, they theoretically provide the freshest data for queries; PG-SR follows the weak session consistency model, which may present different freshness, and we then select it for comparison. We first run *ConsistencyCheck* on static data and demonstrate the freshness. The $freshness(Q)$ should be zero in theory. However, *HATtrick* defines the freshness by the difference between the query start time and the commit time of the earliest invisible transaction in the whole system, and then reports a non-zero freshness value, i.e., 60.4ms, as shown in Fig. 14. We then submit *ConsistencyCheck* on the dynamic data but we configure a long synchronization interval, i.e., 10s. We find the freshness of *HATtrick* is dominated by the synchronization interval, but the commit time of the readable version in *A*-engine and the most recent version in *T*-engine $w.r.t$ current query time differ by a small gap. So the query-oriented method of *Vodka* offers a more practical evaluation of data freshness compared to *HATtrick*.

Next, we launch *ConsistencyCheck* with various TPS scenarios to accurately expose the data freshness of different HTAP systems. We run it for 10 times at ten random time points while running *Vodka* for a duration of 10 mins. The results are summarized in Table 2. With the default configuration, OceanBase and TiDB conform to their linear consistency model by providing the freshest data for queries under various supported TPS, i.e., $freshness(Q) = 0$. Meantime, under a low TPS ($\leq 3.5K$), i.e. a small size synchronization data, PG-SR exposes $freshness(Q) = 0$. However, as the TPS increases, the freshness of PG-SR shows a significant decline. It is caused by its asynchronous synchronization mechanism.

In summary, systems with a linear consistency model can ensure the best freshness, satisfying real-time business needs; in contrast, systems with weak consistency still require in-depth optimization of synchronization capability to improve freshness.

Data Sharing. Since none of the existing HTAP benchmarks propose benchmarking data sharing efficiency, we thus only demonstrate the result of *Vodka*. We present the average data sharing latency by running the point queries to the updated data at 10 random test time points under different throughputs in Fig. 15.


(a) Latency & Cardinality for Q5 (b) Latency & Cardinality for Q8
Figure 10: Controlability of Computation Complexity.

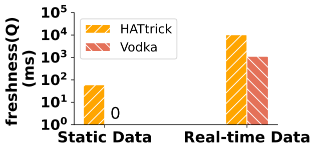


Figure 14: *Vodka* vs *HATtrick* in Freshness on PG-SR

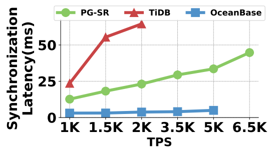


Figure 15: Data Sharing Efficiency

TiDB performs the worst since it has a long synchronization pathway by introducing an additional RocksDB [47] instance (named raftdb) for recording write logs. OceanBase performs the best for its MVCC-based data sharing strategy, which can avoid expensive data migration. For PG-SR, its overall cost is relatively lower than TiDB. This is mainly due to TiDB's complicated synchronization procedure and implementation architecture as introduced before. When TPS increases, the synchronization latency increases obviously for a log-based synchronization model as in TiDB and PG-SR.

In summary, logical isolation on each node (OceanBase) provides more effective data sharing than physical isolation (PG-SR). Simplifying log writing and accelerating log replay is still an opportunity to improve data sharing efficiency.

8.3 Insight Summarization

Resource Isolation. The physical storage isolation can achieve optimal isolation effects for the hybrid workload, but it is infeasible to achieve a global resource scheduling. In contrast, logical isolation may provide adaptive resource configuration, but at the cost of performance interference from resource contention. An adaptive isolation solution is expected for effective resource scheduling.

Consistency Model. A linear consistency model is an optimal choice for the high freshness requirement, but it may degrade query performance for the strict version synchronization. To satisfy queries with various freshness requirements, a better choice is to prioritize the design of a linear consistency model, based on which allows customizing queries with different consistencies.

Data Sharing. The asynchronous synchronization efficiency is closely related to log writing/replay strategies. Parallel log replay and on-demand data synchronization may be preferred. The MVCC-based data sharing requires minimal data migration but needs an effective version management mechanism for version search and garbage collection. Additionally, for a cloud-native HTAP system, it is critical to shorten the pathway for data sharing caused by the common computation and storage separation architecture.

9 RELATED WORK

CH-benCHmark [12] is the first work that successfully addresses the incompatibility between TPC-C and TPC-H, including the scaling models and the schema. Based on *CH-benCHmark*, *HTAPBench* [11] introduces a new testing process that assesses the maximum supported OLAP clients under a specified TPS. However, lack of control of parameter instantiation, neither of them can guarantee the fairness of benchmarking resource isolation, consistency model, and data sharing, i.e., the three key techniques in HTAP systems. *OLxPBench* [23] specifies three practical requirements for HTAP benchmarks: 1) schema semantic consistency requires data modified by *T*-engine to be accessible by *A*-engine; 2) real-time queries

Table 2: Benchmarking Freshness Across Systems

HTAP Systems	*Vodka* (with different TPS) [1K,1.5K,2K,3.5K]	5K	6.5K
PG-SR	0ms	2663.27ms	5208ms
OceanBase	0ms	0ms	✗
TiDB	0ms	✗	✗

The symbol of ✗ means systems cannot support the target TPS.

are designed inside the transactions to simulate a scenario where a user makes a real-time decision; 3) domain specificity is realized by introducing the financial and telecommunication scenarios based on the scenarios of the SmallBank [2] and the TATP [57]. However, *OLxPBench* has no designs for benchmarking freshness and data sharing. *HATtrick* [39] combines SSB with customized TPC-C and contributes a novel visualized metric, i.e., the throughput frontier, to benchmark the isolation ability of HTAP systems; it is also the first work defining freshness, which is the global gap between the query start time and the commit time of the earliest invisible transaction. However, plotting the curve of the throughput frontier is time-consuming, which is difficult for benchmarking [75], and the system-level global snapshot lag time may be inconsistent with the query-level version lag time, which can lead to a wrong freshness for queries. *Hybench* [75] abstracts a new online finance HTAP scenario. To address the issue of having too many dimensions in benchmarking comparison, *Hybench* proposes to combine metrics from different dimensions into an integral *H-Score*. However, *Hybench* still cannot control the parameter instantiation of workloads to achieve a fair comparison.

10 CONCLUSION

In this paper, we introduce *Vodka*, the first comprehensive HTAP benchmark that fully evaluates the three key techniques of HTAP systems in a fair way, i.e., resource isolation, consistency model, and data sharing. We provide in-depth analysis to demonstrate *Vodka*'s effectiveness and efficiency. Moreover, we run extensive experiments on classical HTAP systems, and conclude with novel insights for future research directions.

REFERENCES

[1] Daniel J Abadi, Peter A Boncz, and Stavros Harizopoulos. 2009. Column-oriented database systems. In *VLDB*, Vol. 2. 1664–1665.
[2] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. 2008. The cost of serializability on platforms that use snapshot isolation. In *ICDE*. 576–585.
[3] H Appelgate, M Barr, J Beck, FW Lawvere, FEJ Linton, E Manes, M Tierney, and F Ulmer. 1969. Distributive laws. In *Seminar on Triples and Categorical Homology Theory: ETH 1966/67*. 119–140.
[4] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. In *VLDB*, Vol. 7. 181–192.
[5] Altan Birler. 2019. Scalable Reservoir Sampling on Many-Core CPUs. In *SIGMOD*. 1817–1819.
[6] Mokrane Bouzeghoub. 2004. A Framework for Analysis of Data Freshness. In *IQIS*. 59–67.
[7] 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. In *VLDB*, Vol. 11. 1849–1862.
[8] Jacopo Cavazza and Vittorio Murino. 2016. Active Regression with Adaptive Huber Loss. In *CoRR*, Vol. abs/1606.01568.
[9] 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. In *VLDB*, Vol. 15. 3411–3424.
[10] Byonggon Chun, Jihun Ha, Sewon Oh, Hyunsung Cho, and MyeongGi Jeong. 2019. Kubernetes enhancement for 5G NFV infrastructure. In *ICTC*. 1327–1329.
[11] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2017. Htapbench: Hybrid transactional and analytical processing benchmark. In *ICPE*.

293–304.
[12] Richard Cole, Florian Funke, Leo Giakoumakis, Wey Guy, Alfons Kemper, Stefan Krompass, Harumi Kuno, Raghunath Nambiar, Thomas Neumann, Meikel Poess, et al. 2011. The mixed workload CH-benCHmark. In *DBTest*. 1–6.
[13] Irving M Copi, Carl Cohen, and Kenneth McMahon. 2016. *Introduction to logic.*
[14] Yanlei Diao, Paweł Guzewicz, Ioana Manolescu, and Mirjana Mazuran. 2021. Efficient exploration of interesting aggregates in RDF graphs. In *SIGMOD*. 392–404.
[15] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. In *SIGMOD*, Vol. 40. 45–51.
[16] Lise Getoor, Benjamin Taskar, and Daphne Koller. 2001. Selectivity estimation using probabilistic models. In *SIGMOD*. 461–472.
[17] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. Hyrise: a main memory hybrid storage engine. In *VLDB*, Vol. 4. 105–116.
[18] Aditya Gurajada, Dheren Gala, Fei Zhou, Amit Pathak, and Zhan-Feng Ma. 2018. Btrim: hybrid in-memory database architecture for extreme transaction processing in vldbs. In *VLDB*, Vol. 11. 1889–1901.
[19] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. In *VLDB*, Vol. 15. 752–765.
[20] Wassily Hoeffding. 1994. Probability inequalities for sums of bounded random variables. In *The collected works of Wassily Hoeffding*. 409–426.
[21] Paul W Holland and Roy E Welsch. 1977. Robust regression using iteratively reweighted least-squares. In *Communications in Statistics-theory and Methods*, Vol. 6. 813–827.
[22] 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. In *VLDB*, Vol. 13. 3072–3084.
[23] Guoxin Kang, Lei Wang, Wanling Gao, Fei Tang, and Jianfeng Zhan. 2022. Olxpbench: Real-time, semantically consistent, and domain-specific are essential in benchmarking, designing, and implementing htap systems. In *ICDE*. 1822–1834.
[24] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. 195–206.
[25] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. In *VLDB*, Vol. 11. 2209–2222.
[26] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making mvcc systems htap-friendly. In *SIGMOD*. 49–64.
[27] Kyoungmin Kim, Sangoh Lee, Injung Kim, and Wook-Shin Han. 2024. Asm: Harmonizing autoregressive model, sampling, and multi-dimensional statistics merging for cardinality estimation. In *SIGMOD*, Vol. 2. 1–27.
[28] 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 *ICDE*. 1253–1258.
[29] Sophie Lambert-Lacroix and Laurent Zwald. 2011. Robust regression through the Huber's criterion and adaptive lasso penalty. In *Electronic Journal of Statistics*, Vol. 5. 1015–1053.
[30] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. In *VLDB*, Vol. 8. 1740–1751.
[31] 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. In *VLDB*, Vol. 10. 1598–1609.
[32] Guoliang Li and Chao Zhang. 2022. HTAP databases: What is new and what is next. In *SIGMOD*. 2483–2488.
[33] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: Generating Enormous Query-Aware Test Databases. In *ATC*. 575–586.
[34] Eric Lo, Nick Cheng, Wilfred WK Lin, Wing-Kai Hon, and Byron Choi. 2014. MyBenchmark: generating databases for query workloads. In *VLDBJ*, Vol. 23. 895–913.
[35] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *SIGMOD*. 2530–2542.
[36] Stephen Macke, Maryam Aliakbarpour, Ilias Diakonikolas, Aditya Parameswaran, and Ronitt Rubinfeld. 2021. Rapid approximate aggregation with distribution-sensitive interval guarantees. In *ICDE*. 1703–1714.
[37] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In *SIGMOD*. 37–50.
[38] Sergey Melnik, Atul Adya, and Philip A Bernstein. 2008. Compiling mappings to bridge applications and databases. In *TODS*, Vol. 33. 1–50.
[39] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In *SIGMOD*. 1810–1824.
[40] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-efficient aggregation: Hashing is sorting. In *SIGMOD*. 1123–1136.
[41] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedon Chen. 2007. The star schema benchmark (SSB). In *Pat*, Vol. 200. 50.
[42] Oracle. 2024. Real-Time Access to Real-Time Information. https://www.hunkler.de/files/downloads/oracle-wp-golden-gate-12c-real-t.pdf./.
[43] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. In *Gartner*. 4–20.
[44] PostgreSQL. 2024. Streaming Replication Protocol. https://www.postgresql.org/docs/current/protocol-replication.html.
[45] 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. In *VLDB*, Vol. 6. 1080–1091.
[46] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In *SIGMOD*. 2043–2054.
[47] RocksDB. 2024. https://rocksdb.org.
[48] Stuart J Russell and Peter Norvig. 2010. *Artificial intelligence a modern approach.*
[49] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. In *EDBT*. 540–551.
[50] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. In *VLDB*, Vol. 17. 3290–3303.
[51] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In *OSDI*. 219–238.
[52] Dong Keun Shin and Arnold Charles Meltzer. 1994. A new join algorithm. In *SIGMOD*, Vol. 23. 13–20.
[53] Niharika Singh and Ashutosh Kumar Singh. 2018. Data privacy protection mechanisms in cloud. In *Data Science and Engineering*, Vol. 3. 24–39.
[54] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance characterization of HTAP workloads. In *ICDE*. 1829–1834.
[55] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. In *VLDBJ*. 1–31.
[56] Wei Chit Tan. 2018. Efficient Query Reverse Engineering for Joins and OLAP-Style Aggregations. In *APWeb/WAIM (2)*. 53–62.
[57] TATP. 2009. TATP Benchmark. https://tatpbenchmark.sourceforge.net/.
[58] TiDB. 2024. Hybrid Deployment Topology. https://docs.pingcap.com/tidb/stable/hybrid-deployment-topology.
[59] TPC. 1999. TPC-H Benchmark. http://www.tpc.org/tpch/.
[60] TPC-DS. 1999. TPC-DS Benchmark. http://www.tpc.org/tpcds/.
[61] 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 *SIGMOD*. 789–796.
[62] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. 2022. Near-data processing in database systems on native computational storage under htap workloads. In *VLDB*, Vol. 15. 1991–2004.
[63] Vodka. 2024. Technical Report And Source Codes Of Vodka. https://anonymous.4open.science/r/vodka-8E6B/.
[64] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, et al. 2023. Polardb-imci: A cloud-native htap database system at alibaba. In *SIGMOD*, Vol. 1. 1–25.
[65] Qingshuai Wang, Hao Li, Zirui Hu, Rong Zhang, Chengcheng Yang, Peng Cai, Xuan Zhou, and Aoying Zhou. 2024. Mirage: Generating Enormous Databases for Complex Workloads. In *ICDE*. 3989–4001.
[66] Qichen Wang, Qiyao Luo, and Yilei Wang. 2024. Relational Algorithms for Top-k Query Evaluation. In *SIGMOD*, Vol. 2. 1–27.
[67] Wikipedia. 2024. Bayes' theorem. https://en.wikipedia.org/.
[68] Wikipedia. 2024. Order statistic tree. https://en.wikipedia.org/wiki/.
[69] Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. 2023. Communication-Optimal Parallel Reservoir Sampling. In *BTW (LNI, Vol. P-331)*. 567–578.
[70] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. In *VLDB*, Vol. 10. 781–792.
[71] Quanqing Xu, Chuanhui Yang, and Aoying Zhou. 2024. Native Distributed Databases: Problems, Challenges and Opportunities. In *VLDB*, Vol. 17. 4217–4220.
[72] Siyu Yan, Xiaoliang Wang, Xiaolong Zheng, Yinben Xia, Derui Liu, and Weishan Deng. 2021. ACC: Automatic ECN tuning for high-speed datacenter networks.

In *SIGCOMM*. 384–397.

[73] 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. In *VLDB*, Vol. 13. 3313–3325.

[74] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. In *VLDB*, Vol. 16. VLDB Endowment, 3728–3740.

[75] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. In *VLDB*, Vol. 17. 939–951.

[76] Huidong Zhang, Luyi Qu, Qingshuai Wang, Rong Zhang, Peng Cai, Quanqing Xu, Zhifeng Yang, and Chuanhui Yang. 2023. Dike: A benchmark suite for distributed transactional databases. In *SIGMOD*. 95–98.