Vodka: Rethink Benchmarking Philosophy in HTAP Systems

Zirui Hu
East China Normal University
zrhu@stu.ecnu.edu.cn

Siyang Weng
East China Normal University
syweng@stu.ecnu.edu.cn

Zhicheng Pan
East China Normal University
zcpan@stu.ecnu.edu.cn

Rong Zhang
East China Normal University
rzhang@dase.ecnu.edu.cn

Chengcheng Yang
East China Normal University
ccyang@dase.ecnu.edu.cn

Peng Cai
East China Normal University
pcai@dase.ecnu.edu.cn

Xuan Zhou
East China Normal University
xzhou@dase.ecnu.edu.cn

Quanqing Xu
OceanBase, Ant Group
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
OceanBase, Ant Group
rizhao.ych@oceanbase.com

ABSTRACT

For real-time analysis of up-to-date data, hybrid transaction/analytical processing (HTAP) systems have been extensively studied. In general, three techniques play a critical role in HTAP systems, which are resource isolation, consistency model, and data sharing. However, there still lacks a benchmark suite that could comprehensively cover the three 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). In this paper, we propose *Vodka* to address the above challenges. For resource isolation, we formalize the change of query cardinalities 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 column value 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 system developments.

PVLDB Reference Format:
Zirui Hu, Siyang Weng, Zhicheng Pan, Rong Zhang, Chengcheng Yang, Peng Cai, Xuan Zhou, Quanqing Xu, and Chuanhui Yang. Vodka: Rethink Benchmarking Philosophy in HTAP Systems. PVLDB, 19(3): 481 - 494, 2025.
doi:10.14778/3778092.3778107

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBHammer/Vodka-2025.

1 INTRODUCTION

In general, DBMSs can be broadly classified into two categories according to their functionality, i.e., online transactional processing (OLTP) and online analytical processing (OLAP) systems. With the increasing demand for real-time analytics, such as fraud detection and stock price monitoring, many database vendors [24, 27, 31, 33, 77] propose to combine OLTP ($T-$) and OLAP ($A-$) engines to construct a new type of system, named as hybrid transactional and analytical processing (HTAP) system [48].

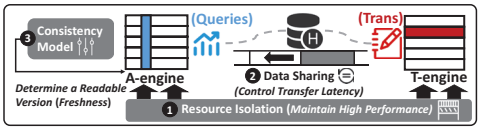


Figure 1: Key Techniques in HTAP Systems

However, there is no simple way to seamlessly integrate the $T-$ and $A-$ engines into a single system, as they might have conflicting system design principles [47, 61]. Therefore, in addition to the design targets mentioned above, the HTAP systems (HTAPs) generally apply three key techniques, which are *resource isolation*, *consistency model*, and *data sharing*. Specifically, as illustrated in Fig. 1, HTAPs try to connect two engines in a real-time manner. ❶ 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. ❷ 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 also uses ❸ 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 simultaneously guaranteed in any HTAP system. For example, placing the two engines on separate nodes with hardware isolation could absolutely eliminate resource interferences, but complicate the efficiency of data sharing and might further decrease data freshness [24]. As a result, the community has reached a consensus that HTAPs need to strike a trade-off between the design decisions of the three techniques [61]. Recently, HTAP benchmarks have been extensively studied. However, providing a comprehensive evaluation of the three critical techniques in HTAPs has always been tough work. The core challenges come from the requirements of:

Consistent Workload Resource Consumptions (C1). The resource isolation technique ensures that integrating OLTP and OLAP businesses into one system does not cause evident performance

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778107

interferences or degradations [52]. 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) [42, 46, 61]. Previous benchmark studies propose to use heuristic methods [11, 78] 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 during benchmarking. This is because different systems might have various OLTP throughputs, leading to different data size increasing rates in both engines. Moreover, even under the same OLTP throughput, 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 dynamically change over time, this implies that the cardinality of each query operator would change indeterminately [40]. Therefore, it's not a fair comparison between different HTAPs when they face workloads with inconsistent computational complexities and resource demands.

Query-oriented Freshness Evaluation (C2). The consistency model determines the gap in data versions between the *T*- and *A*-engines, typically evaluated using data freshness. However, most of the existing work lacks an effective method for freshness calculations [44]. Although *HATtrick* [44] defines 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 (i.e., t_{s_1}-t_{c_2} in Fig. 2). However, it has two drawbacks. Firstly, users are more concerned with whether they could visit the most recent version of data within the specific query range [4, 68]. The data outside the query range does not contribute to its online analytics. 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. 2, the *T*-engine maintains version v_3, while the *A*-engine maintains version v_1 whose commit time is 1 second ahead of v_3. Then, 10 minutes later at t_{s_1}, suppose a query on the *A*-engine could still access v_1. If the freshness is calculated based on the query start time, it indicates that the data is extremely stale. However, even if an analytical query is performed on the *T*-engine, the data being analyzed is only 1 second later than observed in the *A*-engine, which is unlikely to cause a significant analytical bias. It only indicates that data is not synchronized promptly, rather than reflecting true staleness. Thus, the freshness should be calculated as t_{c_3}-t_{c_1}.

Precise Data Sharing Efficiency Measurement (C3). The data sharing process helps transfer data from the *T*-engine to the *A*-engine, generally measured by the synchronization latency. Note, data freshness reflects the real-time difference in data versions between the two engines, while data sharing efficiency measures the duration required for data to be transferred from the *T*-engine to the *A*-engine. For instance, some HTAPs with high consistencies could always achieve high data freshness. However, in these systems, queries usually have to wait for the data transfer to be completed before returning query results. As a result, their poor

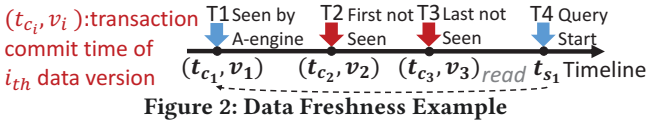


Figure 2: Data Freshness Example

OLAP performance might be caused by the low data sharing efficiency [16, 24, 77], which is irrelevant to intensive resource contentions described above. Thus, HTAPs [5, 76] strive to optimize data sharing efficiency. However, the precise measurement of synchronization latency is still vacant in existing studies. The main challenge is that the system's real-time synchronization status is invisible in a black-box mode. Moreover, instrumenting the kernels to catch the internal synchronization state is laborious or even impossible, especially for closed-source or cloud-native databases.

To address the above challenges, we propose a comprehensive benchmarking philosophy and implement a practical toolkit *Vodka*. First, to address **C1**, we propose to 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 explore 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. Second, to address **C2**, we propose a new definition of freshness for different consistency models. It specifies the differences between the latest data version written in the *T*-engine and the data version being queried in the *A*-engine. We then propose a decoupled version tracking approach to measure it efficiently. On the server side, we attach a version column to each table to track tuple-level evolution. Meanwhile, the client side maintains the actual evolution of each column value, thereby reducing overhead for both *T*-and *A*-engines. In this way, we can compare the latest data versions stored on the client side with the accessed versions by the query to measure the data freshness, thereby avoiding accessing the *T*-engine for acquiring such information. Finally, we adopt a point query driven mechanism to tackle **C3**. By continuously issuing a lightweight point query to retrieve the last committed write, we can confirm whether all preceding writes are visible in the *A*-engine, thereby swiftly detecting synchronization completion time.

In summary, we make the following contributions:

- We are the first approach to propose a comprehensive HTAP benchmarking philosophy addressing the challenges C1-C3.
- We implement an open-sourced benchmark suite [69] following the above philosophy.
- We launch extensive experiments over three typical systems with our benchmarking suite, and provide novel insights for future HTAP system developments.

2 BACKGROUND

Three key techniques are widely adopted in HTAPs [61].

Resource Isolation. HTAPs employ resource isolation strategies to schedule and separate the hardware resources for mitigating resource contentions between *T*- and *A*-engines. ❶ *Unified Resource.* The *T*- and *A*-engines share all resources of the same node, and they use logical isolation methods to schedule resources [52]. Specifically, for the CPU resources, the virtualization technique can be applied

to split resources into groups and assign them to T- and A-engines separately according to their predefined usage limits [41]. Another way is to bind CPUs to different NUMA nodes [42, 64]. For memory resources, the virtualization technique can also be applied [41], or maintaining two in-memory data replicas to serve OLAP reads and OLTP writes, respectively [16, 18, 31]. ❷ *Decoupled Resource.* The T- and A-engines are deployed on different nodes. For memory isolation, each of the T- and A-engines could use all the CPU and memory resources of their assigned nodes [5, 67], but they need to share all underlying storage resources. For disk isolation, T- and A-engines have their dedicated CPU, memory, and storage resources through absolute hardware isolations [24, 76].

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, resulting in various degrees of data freshness. Generally, the specific version gap is regulated by consistency models. ❶ *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 [31], OceanBase [77], and TiDB [24]. ❷ *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 [7], BatchDB [42], and Vegito [57] are the representative systems. ❸ *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 [5], Aurora[67] and HyPer[27].

Data Sharing. It refers to the method of sharing the produced fresh data from the T-engine to A-engine for consumption [60]. ❶ *Share Everything.* The T- and A-engines share a single copy of data in the storage layer. With the help of MVCC mechanism [73], it allows the A-engine to access multi-versions of records created by the T-engine. However, the A-engine might need high overhead in traversing version chains to find the expected versions [28]. ❷ *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 [27]. Specifically, when each query arrives, it creates a virtual in-memory data snapshot for subsequent reading. For new OLTP updates, it employs a Copy-on-Write mechanism to write to a new memory location. Thus, it results in high memory usage in write-intensive scenarios [35]. Another approach is to maintain an 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, it causes additional memory consumption. ❸ *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. 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 costs in log shipping and replaying [24, 76].

3 VODKA FRAMEWORK

The Fig. 3 shows the framework of *Vodka* that has three components.

Resource Isolation. Users are primarily concerned about whether the HTAP system could mitigate resource contentions well, which guarantees optimal OLAP performance while satisfying a specific

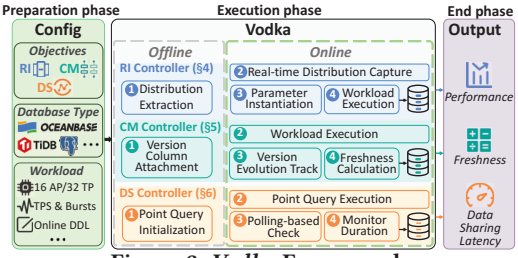


Figure 3: *Vodka* Framework

OLTP throughput requirement [42, 61]. 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 trend, enabling the same data growth rate across systems [10, 56]. However, the query complexity in the A-engine is still inconsistent due to different query start timepoints and data sizes. 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, ensuring that the cardinalities of query operators increase linearly with the data scale. Note, this cardinality control applies to both synthetic and real-world datasets; 3) construct a regression model to deduce the query latency of each A-engine according to the underlying data size, which enables performance comparison across A-engines even under different data sizes. Note, to reflect the real-world variability, the OLTP throughput or read-/write transaction ratios can be adaptively adjusted.

Consistency Model. We compute freshness by the differences in commit time between the latest data version written in the T-engine and the data version being observed in the A-engine. Though the internal MVCC version chain provides an opportunity to trace data versions, it is inaccessible to users. Thus, we propose a lightweight version tracking method to trace version evolutions. The main idea is to attach a new version column on each table to track the tuple-level version evolution on the server side. The detailed column data evolution time, along with the column-level version values, are stored on the client side. Then, based on the queried data returned from the A-engine, we can accurately locate and track the corresponding data evolutions on the client side, which also triggers the client-side garbage collection for stale versions. Finally, the freshness is calculated as the maximum commit timestamp gap between two engines regarding all the query accessed data items.

Data Sharing. To accurately measure synchronization latency in black-box systems where the internal synchronization state is invisible, 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 HTAPs, the synchronization status of this last committed write guarantees that all prior writes are synchronized. Note, the attached version column to each table indicates the version evolution for the latest write. We conduct a point query to check the version consistency of this last committed write in the A-engine, repeating this until the consistency between two engines is confirmed. The time taken to achieve this consistency is the data sharing latency. Given the rapid response of the point query, the synchronization completion can be captured in time.

Running Example. After users set the configurations, *Vodka* initiates the benchmarking process. *For resource isolation evaluation*, ❶ In the offline stage, it acquires data distribution information by reusing user-specified distribution functions for some specific attributes, or by performing static sampling for some attributes without user specifications. ❷ In the online stage, it applies reservoir sampling to capture dynamic changes in both single-attribute and multi-attribute joint distributions. ❸ The transactions are generated according to user-defined configurations (e.g., dynamic workload patterns), while the queries are instantiated at runtime using a distribution-guided cardinality control method. ❹ Finally, *Vodka* executes the hybrid workloads. *For consistency model evaluation*, ❶ In the offline stage, it attaches a version column to each table that helps track the tuple-level version evolution on the server side. ❷ In the online stage, it generates HTAP workloads according to the configuration file and executes. ❸ On the client side, it tracks the version evolution of data items that are modified by transactions in the T-engine, and retrieves the version information of the data touched by OLAP queries in the A-engine. ❹ Finally, it computes the freshness metric according to the previous definition. *For data sharing evaluation*, ❶ In the offline stage, it prepares a lightweight point-query template for detecting the visibility of the latest committed write. ❷ In the online stage, it embeds these point queries into the user-configured HTAP workloads and executes them in real time. ❸ On the client side, it issues polling-based point queries with high priority to the A-engine to check whether the data committed before a given time has been synchronized. ❹ Finally, it determines the data sharing latency by recording the duration when the latest committed write in the T-engine becomes visible in the A-engine.

4 BENCHMARK RESOURCE ISOLATION

4.1 Query Cardinality Formalization

Consider a query q, its logical execution plan can be represented by a query tree. Each leaf node in the 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 selectivity $Sel(\sigma_P(R_i))$ of a selection operator is defined as the proportion of tuples in R_i satisfying 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., $Pr(\sigma_P(R_i))$. 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 is the probability that both the selection and join predicates are satisfied simultaneously in the value space $R_i \times R_j$.

However, in real-world scenarios, there might exist intricate dependencies between columns. Although cardinality estimation has been extensively studied, it is still challenging to derive accurate cardinality under arbitrary query templates and parameters [19, 23, 29]. In light of this, we propose to follow the previous OLAP benchmarks [45, 65, 66] 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 independence 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).

$$Pr(\sigma_P(R_i)|\sigma_P(R_j))=Pr(\sigma_P(R_i)),\ Pr(\sigma_P(R_j)|\sigma_P(R_i))=Pr(\sigma_P(R_j)) \quad (1)$$

$$Pr(\sigma_P(R_i)|R_i \bowtie_J R_j)=Pr(\sigma_P(R_i)),\ Pr(\sigma_P(R_j)|R_i \bowtie_J R_j)=Pr(\sigma_P(R_j)) \quad (2)$$

Based on the Bayes' theorem [72], 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).

$$Sel(\sigma_P(R_i) \bowtie_J \sigma_P(R_j)) = Pr(\sigma_P(R_i), \sigma_P(R_j), R_i \bowtie_J R_j) \quad (3)$$

$$= Pr(\sigma_P(R_i), \sigma_P(R_j)|R_i \bowtie_J R_j) \cdot Pr(R_i \bowtie_J R_j) \quad (4)$$

$$= Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot Pr(R_i \bowtie_J R_j) \quad (5)$$

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.

$$Card(\sigma_P(R_i) \bowtie_J \sigma_P(R_j))=Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot Pr(R_i \bowtie_J R_j) \cdot |R_i| \cdot |R_j|$$
$$=Pr(\sigma_P(R_i)) \cdot Pr(\sigma_P(R_j)) \cdot |R_i \bowtie_J R_j| \quad (6)$$

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

Next, we discuss the cardinality control of selection and join operators, and put the discussion of the projection and aggregation operators in our technical report [69] due to space constraints.

4.2 Selection Selectivity Control

The two widely used query parameter instantiation strategies for selectivity control in previous work [11, 44] are random and fixed parameter instantiations. However, they have specific limitations. Take the query Q14 in TPC-H as an example. As shown in Fig. 4, it consists of a selection operator containing the predicate $t_1 \geq p_1 \wedge t_1 \leq p_2$, a join operator on the tables ORDERLINE and ITEM, and an aggregation operator. Here, t_1 is a timestamp attribute that increases monotonically, and p_1 and p_2 are two parameters that need to be instantiated during benchmarking. We perform continuous data updates over 30 seconds. If we take the fixed parameters instantiation strategy, we observe that it would keep operator cardinalities that should evolve with the new data modifications remain unchanged, and prevent access to fresh data. In contrast, the random parameters instantiation causes unpredictable cardinality fluctuations, making query complexity beyond control. Therefore, we need to dynamically instantiate the query parameters more reasonably.

The main idea of selection selectivity control in *Vodka* 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 the underlying tables. 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 attributes or joint distribution of arithmetic functions operating on multiple attributes, where these

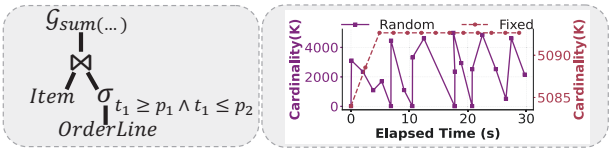


Figure 4: An Example of Inconsistent Query Complexity

distributions could change over time [10, 11]. To address this issue, we propose to use the reservoir sampling [3] to continuously track the distribution changes by randomly choosing a batch of samples in only one pass over the streaming data. Then, we can use the small batch of sampled data to approximate real data distributions, which is further leveraged to help instantiate query parameters and perform an online cardinality calculation. 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 parameters in the selection predicate P according to the data distributions. For ease of presentation, we mainly discuss the case in which P follows the conjunctive normal form (CNF), i.e., $P = \wedge_{x=1}^{n} clause_x \; s.t. \; clause_x = \vee_{y=1}^{m} literal_{xy}$. Note, any other form of predicate can be transformed to CNF [23, 25, 55]. 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 $g()$. Now, we discuss the selectivity control of a predicate with various numbers of clauses and literals.

Case 1: n=1 and m=1. Predicate P has only one literal. For the unary predicate, it can be classified into two cases. The first case includes the operators of $<$, $>$, $\leq$, $\geq$, 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 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 is the probability of a row in R_i whose attribute A_k takes a value $\leq l_k$. The second case includes the operators of $=$, $\neq$, (not) *in*, whose predicate seeks to filter out specific values which meet the requirement of selectivity α. For example, considering 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 that is 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 R_i. For this case, we use $g()$ to precompute and dynamically update the result data distribution based on the data distributions of involved columns. Then, we instantiate the parameter according to the result data distribution.

Case 2: n>1 and m=1. Predicate P has n clauses, each of which has one literal. Suppose that 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 [20], 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 data distributions of each group, we can find valid parameter values that make the selectivity of each clause equal to $\sqrt[n]{\alpha}$.

Case 3: n=1 and m>1. Predicate P has one clause that contains more than one literal. Then, De Morgan's Law [12] can convert it to case 2 by performing a negation on P.

Case 4: n>1 and m>1. Predicate P 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 two rules in the area of set theory to help simplify P. Specifically, $rule_1$ is $\sigma_{literal_i} \cup \sigma_{literal_j}$=$\sigma_{literal_j}$ if $\sigma_{literal_i} \leftarrow \emptyset$, and $rule_2$ is $\sigma_{clause_i} \cap \sigma_{clause_j}$=$\sigma_{clause_j}$ if $\sigma_{clause_i} \leftarrow U$. 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 empty set and universal set. Then, we can successfully reduce the complex predicates into the above three simple cases.

4.3 Join Cardinality Control

We next discuss how to ensure that the join cardinality 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 [63, 66], we mainly focus on the cardinality control of the PK-FK join. We assume that R_i is the referenced table containing primary keys, and R_j is the referencing table containing foreign keys. 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 pairs of rows in the join input, while a join distinct cardinality requires that there exactly exist n_{jdc} distinct primary/foreign key values in n_{jc} matched pairs. For example, the output size of an equi-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 generally represented as a linear combination of n_{jc} and n_{jdc} [25, 71]. Therefore, in this section, we discuss the cardinality change of n_{jc} and n_{jdc} using equi-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 follows the form:

$$|R_i' \bowtie_J R_j'| = (c_1 \cdot \frac{|R_i'|}{|R_i|} + c_2 \cdot \frac{|R_j'|}{|R_j|} + c_3) \cdot |R_i \bowtie_J R_j|. \tag{8}$$

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. 9, we further classify the join scenario into four cases according to whether there exists delta data in each joined table.

$$\begin{aligned} &|R_i' \bowtie_J R_j'| = |(R_i \cup \Delta R_i) \bowtie_J (R_j \cup \Delta R_j)| \\ &=|(R_i \bowtie_J R_j) \cup (\Delta R_i \bowtie_J R_j) \cup (R_i \bowtie_J \Delta R_j) \cup (\Delta R_i \bowtie_J \Delta R_j)| \end{aligned} \tag{9}$$

Case 1: ΔR_i=$\emptyset$ and ΔR_j=$\emptyset$. This indicates that neither of the joined tables involves delta data. Thus, c_1=0, c_2=0, and c_3=1 in Eq. 8.

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 PKs in R_i due to the referential integrity constraint [43]. Since R_j does not have delta data in this case, we can derive that ΔR_i only contains newly inserted PKs. 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. 8.

Case 3: $\Delta R_i = \emptyset$ 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 equi-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 equi-join is $|R'_j|$. That is, c_1=0, c_2=1 and c_3=0 in Eq. 8. For semi-join, as the *FK* values usually follow a user-specified distribution when *PK*s are fixed [20, 66], we propose to keep modifications of *FK* values comply with the specific distribution. With this method, we can infer that $R_i \ltimes \Delta R_j$ 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. 8.

Case 4: $\Delta R_i \neq \emptyset$ and $\Delta R_j \neq \emptyset$. This indicates that both tables have delta data participating in the joining process. In this case, to ensure the non-decreasing cardinality size of each join operator when scaling the data size, previous studies [11, 44, 78] do not perform any delete operation on join tables. This is because delete operations 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 follow these studies and use the *distributive law* [1] to deduce the join result in Eq. 10.

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

As there exist new *PK* values inserted into the referenced table R_i and these *PK* values would also participate 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 *PK*s in R_i and ${\Delta R_j}^{new}$ references the new *PK*s in R_i. Then, we can further transform $(R_i \cup \Delta R_i) \bowtie_J \Delta R_j$ into Eq. 11. For equi-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. 8.

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

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 $\theta_1 = \frac{|R_i \ltimes R_j|}{|R_i|}$, which indicates that the proportion keeps unchanged even after performing modifications on both of the two joined tables. 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. 8. Note, this control of PK-FK joins can be extended to support FK-FK/non-equi joins. The main idea is to guarantee that the newly produced join cardinality from the delta data scales proportionally with the original join cardinality. The specific scale factor can be adaptively adjusted according to users' requirements. We put the technical details in our technical report [69].

4.4 Performance Measurement

As previously discussed, the query complexity of each *A*-engine would be inconsistent since different query start times lead to different table sizes and cardinalities. Since we have ensured that the cardinality of each query operator scales linearly with 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. Note, *Vodka* does not aim to build a universal query performance prediction model for arbitrary queries, while we adopt a posterior approach to fit the performance curve for each query as the data scale changes. Moreover, previous learning-based studies (e.g., ALEX [14] and PGM [17]) have proved that a fine-grained segmentation can help to achieve accurate local fitting. Thus, we also apply a piecewise latency fitting method. Specifically, it divides the workload into a certain number of segments and then fits the performance for different segments.

Moreover, as the *A*-engine has employed many optimization techniques to handle large-scale queries [8, 62, 75], the time complexity of each query operator is generally $O(M)$ 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 $O(MlogM + NlogN)$ [30, 58]. Considering that each query can be represented as a query tree, then the output of one operator serves as the input for the subsequent operator [40]. Since we have guaranteed that the output size of each query operator increases linearly with the table size, it motivates us to build a 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 [6, 32] 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 data versions. On this basis, we describe three consistency models.

Version Write. The write operation $w_i(x, val_i)$ denotes a transaction committed at t_i that writes data item x with the value val_i.

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.

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 $S_t(x)$, i.e., $|S_t(x)|$.

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 $CM_{TA}(w(x, val_l), r_{s_1}(x)) = |S_{t_l}(x)| - |S_{t_{s_1}}(x)| = g_1$, $CM_{TA}(w(x, val_l), r_{s_2}(x)) = |S_{t_l}(x)| - |S_{t_{s_2}}(x)| = g_2$, $CM_{AA}(r_{s_1}(x), r_{s_2}(x)) = |S_{t_{s_1}}(x)| - |S_{t_{s_2}}(x)| = g_3$.

Linear Consistency refers to the data version accessed on the A-engine exactly equals to the latest data version in the T-engine. In this case, we require g_1=g_2=g_3=0. **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 (may not be the latest data version in the T-engine). That is, g_1=$g_2 \geq 0$ and g_3=0. **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 $g_2 \geq 0$.

According to the consistency model definitions, the measurement of freshness should precisely reflect the degree of version gap. Thus, 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. Note, for some new inserts that have never been visible in the analytical query's result, we could only acquire information regarding the time point of the tuple creation. In this case, we switch to using the query start time and version creation time to calculate the freshness. Further, the freshness of a query Q launched at t_Q^s is defined as the maximum timestamp gap for all the query's accessed data items. Note, since *Vodka* is an end-to-end benchmark, its freshness metric has implicitly contained the influence of internal processing mechanisms, such as query scheduling delays or stale cache results.

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 compute Q's freshness based on the query results on the two engines. However, this approach incurs additional query execution costs in the T-engine. Moreover, it also poses a challenge in obtaining the latest commit time of each result data item 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 is scanning transaction logs to obtain the timestamp of each data version. Nevertheless, it would result in considerable IO costs and degrade the system performance. Therefore, to minimize the impact on the system performance, a non-intrusive method is to log the whole version evolution 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. Because a value

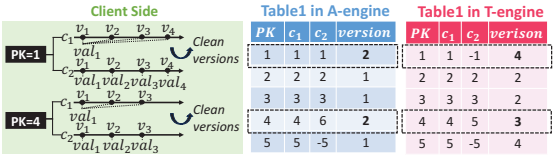


Figure 5: Example of Freshness Measurement

might exist multiple times in the version evolution of a data item, while the query result only contains values, it is ambiguous which version is touched by the query. This situation would cause an inaccurate calculation of freshness. Thus, we need to attach the version information for all data items in the query result.

To this end, the intuitive method 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 [4, 52], while the version evolution of a tuple does not indicate that all of its attribute values are changed. However, attaching version columns also incurs 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 T- and A-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, it maintains the actual version evolution of each column value and its commit timestamp of the version, along with the evolution of its corresponding tuple. For example, in Fig. 5, suppose that query Q accesses c_2 in two tuples with PK=1 or 4. 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, freshness is calculated as $max(ts_4^1 - ts_2^1, ts_3^4 - ts_2^4)$.

Note, the commit timestamp is the client-side time when the transaction successfully returns to the client. Since network latency is negligible compared to the operation latency when the client and server are co-located in the same cluster [36, 37, 39], the client-side commit timestamp can thus be treated as the system's actual commit timestamp. If clients are deployed on a single node, we use its hardware clock time to record timestamps. Otherwise, we use a centralized logical timestamp oracle to allocate logical timestamps and provide a unified global clock time without clock skew [24, 77].

The client side usually maintains the version evolution for a small amount of recently updated column values. First, we only track the version evolution for columns both written by transactions in the T-engine and then accessed by queries to benchmark freshness 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 synchronized to the A-engine, and all the older versions would not help the measurement freshness. In summary, *Vodka*'s memory overhead is mainly affected by the OLTP throughput (high

speed of version creation), the efficiency of the system's internal synchronization (delayed visibility of new versions), and the query volume (delayed acquisition of the version visibility information from the *A*-engine). However, given that the garbage collection is in-memory, it is much faster than the operation of creating new versions in the system, which requires write I/Os. Thus, the old versions could be cleaned faster than the version generation. Considering that the synchronization delay and query latency [57] are generally not very long in HTAPs, *Vodka* can thus be viewed as only tracking the recently written data versions, which is irrelevant to the data size. Moreover, the version management and freshness metric computation are fully parallelizable and can be partitioned and deployed on multiple nodes. Note, sampling or sketching could mitigate this overhead, but at the expense of freshness accuracy since some version evolution information is missing.

6 BENCHMARK DATA SHARING

Users are usually concerned with the real-time synchronization status of the HTAP system when performing their analytic tasks [61, 80]. 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 check whether or when the data written before a certain time point is fully synchronized to the *A*-engine [7, 57, 70]. This inspires us to design a method that measures the synchronization latency of the system at different time points. Specifically, we first randomly select several test time points. 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. The data sharing capacity of the system 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 new data generation is mainly determined by the throughput and write ratio in the *T*-engine. We thus regulate each *T*-engine to exhibit the same throughput trend and workload semantics such that each system experiences the same speed of new 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 [52].

However, since the internal synchronization status is invisible to users, it imposes a great challenge of exactly measuring the synchronization latency. The method of instrumenting kernels to catch the internal synchronization state is laborious or even impossible [59]. As the HTAPs generally ensure the linearizability of writes [2, 34, 74, 80], the order of synchronization 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, all prior writes before the test time point have been synchronized.

To this end, we need to address two 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. For the second issue, similar to the method in measuring freshness, we attach a version column to each table to indicate the version evolution of tuples in the two engines. Specifically, each write changes the version 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, 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. Note, this lightweight verification facilitates rapid responses with quite low query latency and is generally not blocked, ensuring that the synchronization completion time can be caught just in time. Moreover, since this query is only sent to the *A*-engine, it is usually unaffected by the high contentions in the *T*-engine. In addition, as this point query is granted high priority through query hints, it is generally not blocked by other OLAP queries in the *A*-engine.

7 EXPERIMENTAL EVALUATION

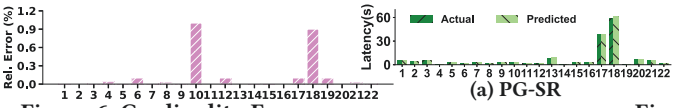
7.1 Experimental Setup

Schema and Workloads. Since TPC-C and TPC-H have been widely used to represent an HTAP scenario [10, 11, 26, 44], we also construct our HTAP scenario from the two benchmarks. We follow *CH-benCHmark* [11] to unify the tables that have almost the same application semantics. To evaluate resource isolation, we make minor adjustments to the TPC-H query templates to be compatible with TPC-C and unify them to generate an HTAP workload. To evaluate consistency model and data sharing, we follow previous work [26, 78] to introduce a dedicated real-time query called *ConsistencyCheck*, defined as SELECT <cols> FROM ORDERLINE WHERE OL_RECEIPTDATE BETWEEN $p1$ AND $p2$ on the ORDERLINE table that undergoes the most frequent writes. Note, $p1$ and $p2$ can be set appropriately such that the latest data can be accessed.

Baselines. The state-of-the-art studies include *HATtrick* [44] and *Hybench* [78]. Both approaches have some similar methods to evaluate resource isolation and consistency models, but they differ in terms of schema and workload design. Since *Vodka* and *HATtrick* construct a similar HTAP scenario based on widely used TPC-C and TPC-H (SSB used in *HATtrick* is a simplified version of TPC-H), we thus choose *HATtrick* as our baseline for comparison.

Cluster and Database Configuration. Each system is deployed on three servers. Each server has an Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz with 96 physical cores, 375GB RAM, and a 1.5TB pmem disk. All nodes are connected via 10GbE network cards. We select three representative HTAP systems in our experiments, which are PG-SR v15.0 [49], OceanBase v4.2.0 [77], and TiDB v7.1.0 [24]. This is because many HTAP systems [9, 41] are closed-source or available as cloud services, which prevents local deployment.

Default Benchmark Configuration. Following [44, 78], all experiments are conducted on a database consisting of `120` warehouses (i.e., `10` GB) by default, and each OLTP thread is bound to one warehouse [51, 52, 79]. When evaluating systems, we determine the OLTP throughput and the number of OLAP threads through preliminary experiments, identifying a proper configuration that



(a) PG-SR (b) OceanBase (c) TiDB

Figure 6: Cardinality Error

Figure 7: Actual and Predicted Latencies

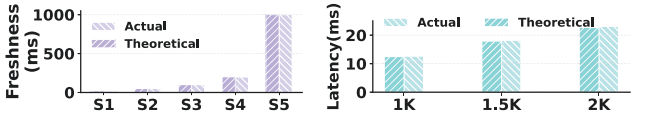
all systems could support. The maximum OLTP throughput is set to 2,000 by default, and the number of OLAP threads is fixed at 10. All experiments have a 5-minute warm-up phase.

7.2 Evaluation of the Key Designs of *Vodka*

7.2.1 Evaluation under Stable Workload Pattern. We first evaluate *Vodka*'s three key techniques under a stable workload pattern.

Accuracy of Cardinality Control and Latency Modeling. We adopt the metrics of average relative error $\frac{1}{|Q_{op}|}\frac{\sum_i ||C_i|-|\hat{C}_i||}{\sum_i |C_i|}$ [38] to measure the accuracy of cardinality control on all operators (Q_{op}) in queries Q, where $|C_i|$ and $|\hat{C}_i|$ are the expected and observe cardinalities of i-th operator. Since our method is not sensitive to the running system, we take PG-SR to illustrate results. Specifically, we run *Vodka* for 5, 10, and 15 minutes, and collect the observed operator cardinalities to calculate the average relative error. Fig. 6 shows the results on all 22 queries. We observe that all queries have low relative errors (maximum error ≈ 1%), which benefit from our effective distribution-guided data manipulation and parameter instantiation method. Next, we measure the accuracy of the regression model in deriving query latencies under changing data sizes. Specifically, we gather 80% pairs of latency and data size for each query as the training set and collect another 20% pairs as the test set. The results are shown in Fig. 7. We observe that the gap between the average actual latency and predicted latency in each system is quite small (maximum error <5%).

Accuracy of Freshness Calculation. We vary the freshness requirement (using 20ms, 50ms, 100ms, 200ms, 1000ms) and report the actual measured freshness result in *Vodka*. Specifically, these requirements are abstracted from five typical real-time analysis scenarios [57], which are fraud detection, online gaming, personalized ads, stock price monitoring, and e-commerce (denoted as S1–S5). Note, the freshness requirement refers to the maximum acceptable delay between the commit timestamp of the data version being read in the *A*-engine (denoted as v_1) and the commit timestamp of the latest version in the *T*-engine (denoted as v_2). During the evaluation process, we set the interval between consecutive transaction commits to be the same as each specified value of the freshness requirement. Then, we set a synchronization delay that forces the system to pause for a specific duration, ensuring that the *A*-engine would retrieve the old version v_1. Finally, we run *Vodka* and report the measured freshness. The results are shown in Fig. 8a. We observe that the result measured by *Vodka* aligns closely with the theoretical freshness values. This is because our novel version tracking approach records data version evolution in detail on the client side, allowing *Vodka* to effectively trace commit times associated with the versions returned by queries.



(a) Freshness Accuracy (b) Data Sharing Latency Accuracy

Figure 8: Evaluate Key Designs under Stable Patterns

Accuracy of Data Sharing Measurement. To evaluate the measurement accuracy of data sharing in *Vodka*, we instrument the kernel of PG-SR to catch the accurate synchronization latency as the baseline [50]. Fig. 8b compares the baseline measurements with those reported by *Vodka*. We observe that *Vodka* also achieves high accuracy in synchronization latency measurement. This is because our lightweight point query driven method could effectively capture the precise synchronization completion time point.

Ablation Study. We finally evaluate the impact of *Vodka*'s design on the system throughput. *Vodka* has three key design features as mentioned in § 4, 5, and 6 (denoted as F1-F3). Fig. 9 shows the results, where each feature is turned on in order. We observe that *Vodka* imposes a quite small overhead on the system performance. This is because most information is collected asynchronously on the client side (e.g., traces from reservoir sampling and data version evolution), which does not impose significant overhead on the server. Moreover, the point query check in the *A*-engine is lightweight, ensuring that the overall throughput is not significantly affected.

7.2.2 Evaluation under Various Workloads. We next evaluate *Vodka*'s three key techniques under various workloads.

Dynamic Workload Patterns. We first generate dynamic workload patterns to evaluate the broad applicability of *Vodka*. For each workload pattern, the throughput is randomly sampled from a range [500, 5000], and the ratio of read/write transactions is randomly selected from the set $\{1:9, 5:5, 9:1\}$. Moreover, the duration of each workload pattern is sampled from a Poisson distribution with $\lambda = 10$, reflecting the real-world non-uniform characteristics of OLTP throughputs, as shown in Fig. 10a. Due to space constraints, we only report the result of PG-SR in subsequent evaluations. The results are shown in Fig. 11 (Default). For resource isolation, the error of operator cardinality control is less than 0.6% under all the 22 queries and workload patterns, and the error of latency fitting is less than 6%. For freshness measurement, it achieves an error of zero under all workload patterns. For data sharing, it has high accuracy of synchronization latency measurement, with the average error less than 1.5%. By default, the following experiments would also be conducted based on these dynamic workload patterns.

Bursts&Online DDLs. Following previous work [22], based on the default dynamic workload patterns in Fig. 10a, we incorporate various online DDLs for TPC-C (e.g., adding columns) and execute them randomly during benchmarking. Moreover, we inject peak loads with OLTP throughputs randomly sampled from a range [6300, 6500] into the workload patterns, where each peak lasts for 10–30s (in Fig. 10b). The experimental results are shown in Fig. 11 (Burst&DDL). The relative errors for the accuracies of cardinality control, query latency fitting, and the measurements of consistency model and data sharing are still low in these scenarios.

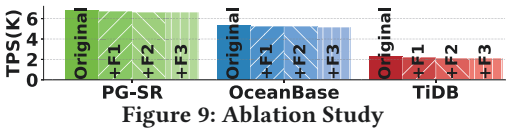
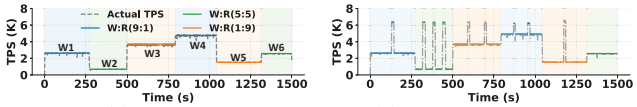
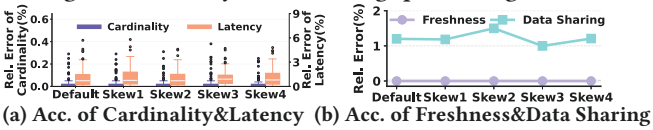


Figure 9: Ablation Study



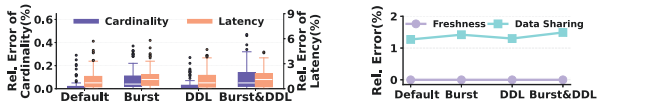
(a) Default (b) Adding Burst&DDL

Figure 10: The Dynamic Throughput Configurations



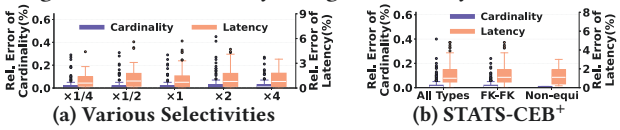
(a) Acc. of Cardinality&Latency (b) Acc. of Freshness&Data Sharing

Figure 12: Evaluate Key Designs under Various Distributions



(a) Acc. of Cardinality&Latency (b) Acc. of Freshness&Data Sharing

Figure 11: Evaluate Key Designs under Dynamic Patterns



(a) Various Selectivities (b) STATS-CEB$^+$

Figure 13: Sensitivity Analysis of Query Changes

Various Distributions. Following previous work [13], we incorporate various levels of data skewness using five Zipfian values. Then the Zipfian levels of queried attributes are randomly selected from the sets $\{0, 1\}$, $\{0, 1, 2\}$, $\{0, 1, 2, 3\}$, and $\{0, 1, 2, 3, 4\}$, denoted as "Skew1/2/3/4", respectively. Fig. 12 shows the result, where "Default" means the original distribution of *Vodka*. We observe that *Vodka* demonstrates high effectiveness under diverse distributions.

7.3 Sensitivity Analysis

The Varied Query Selectivities. Since the query selectivity could affect the query complexity, we thus adjust the query selectivity to 0.25×, 0.5×, 1×, 2×, and 4× of the default TPC-H selectivities to evaluate the accuracy of *Vodka*'s cardinality control and query latency fitting method. The results in Fig. 13a show that *Vodka* consistently has low errors across various selectivities.

The Real-world Distributions. Given that *STATS-CEB* [20] is the state-of-the-art cardinality estimation benchmark with real-world data distributions, we then extend *STATS-CEB* to build an HTAP scenario by introducing new transactions, called *STATS-CEB*$^+$. The detailed workload descriptions are in the technical report [69]. The results in Fig. 13b (All Types) show that *Vodka* consistently has low relative errors for both cardinality control and query latency fitting.

The Non-PK-FK Join Types. Based on *STATS-CEB* containing PK-FK&FK-FK joins, we add some non-equi joins following its semantics. Results in Fig. 13b show that *Vodka* always has high accuracy of cardinality control and query latency fitting.

Scalability Analysis. Next, we evaluate the effect of dataset size, query volume (i.e., the accessed data size in the *ConsistencyCheck* workload), and the internal synchronization efficiency. Note, all the workloads are constructed based on the dynamic workload patterns with peak write pressures in Fig. 10b. Specifically, we vary the dataset size (scale factor) and query volume to 1×, 2×, 4×, and 8× of the default configurations. In addition, we simulate poor synchronization capability by increasing the synchronization delay to 8s. We have two observations in Fig. 14. Firstly, Fig. 14a shows that increasing the dataset size does not significantly increase the overhead. The reason is that *Vodka* only tracks the version evolution of data items that are modified by transactions in the T-engine, rather than maintaining versions for all data items in the database. Secondly, Fig. 14b and Fig. 14c show that increasing either the query volume or the synchronization delay meets a near-linear overhead increase. This is because a larger query volume or a longer synchronization latency could either postpone the acquisition of the version visibility information from the A-engine on the client side or delay the A-engine's visibility of new versions, both leading to version accumulations on the client side. Moreover, since each data item can be processed independently, versions can be partitioned across multiple nodes for parallel processing. Thus, the overhead can be amortized with distributed version management (over 4 nodes) as shown in Fig. 14b (Opt.) and Fig. 14c (Opt.). Note, a larger query volume only increases the computational cost, but does not affect the version information already maintained on the client side. Thus, it does not affect the accuracy of the freshness.

The Impact of Long Transactions. To evaluate the impact of long transactions on freshness and data sharing latency measurement, we follow [22] to generate long transactions based on the TPC-H queries and execute them in the T-engine. We observe that the correctness of freshness calculation maintains zero error, and the accuracy of data sharing measurement is well guaranteed in Fig. 15 (LongTrx). The reason is that the long transactions only affect the version maintenance inside the DBMS, which generally does not affect *Vodka*'s measurement. We observe in Fig. 14d that it also does not significantly increase the memory consumption and computation overhead. This is because our client-side garbage collection is decoupled from the server-side garbage collection. Once a specific version has been touched by the query in the A-engine, we prune all versions older than that version, regardless of the older versions still retained internally in systems by long transactions.

The Impact of High Contentions. To evaluate the impact of high contentions on the measurement of freshness and data sharing latency, we create high contentions in the T-engine by adjusting the ratio between the thread and warehouse number to 8 [53, 79]. We observe that the measurement of these aspects still maintains high accuracy in Fig. 15 (Contention). This is because queries are only sent to the A-engine and generally granted with high priority.

Effect of Version Column. We add an additional version column to record tuple-level version evolution in each table, which may increase the storage and affect query performance. To evaluate this, we compare the latency of all queries under the dynamic workload patterns (denoted as W1-W6) by retaining and removing the version columns. The results are shown in Fig. 16. We observe that the version column has little impact on the query performance, due to its tiny storage cost compared to the original columns.

Model Comparison. We compare *Vodka*'s regression model with two expressive models, i.e., Gradient Boosting Regressor model (GBR) [15] and the MSCN-based model (MSCN) [21]. We observe that *Vodka*'s regression model achieves a higher fitting accuracy while requiring less fitting time in Fig. 17, due to the well-controlled cardinality and the lightweight piecewise fitting mechanism.

7.4 Comparing With Existing Benchmarks

We first compare *Vodka* with *HATtrick* [44] regarding to resource isolations. We propose to measure how query cardinalities and runtimes in *Vodka* and *HATtrick* change under two representative

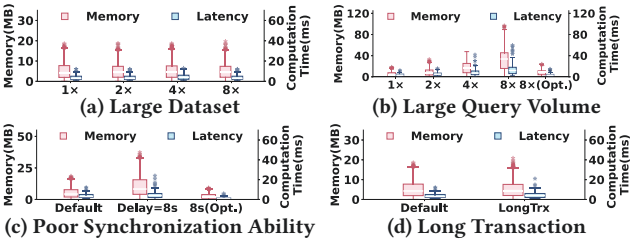
multi-table join queries, i.e., Q5 and Q8. Fig. 19 shows the results. We observe that *HATtrick* exhibits an approximately constant query latency. This is because it always uses fixed query parameters, so queries could thus only access static historical data, even though the data is dynamically increasing. Therefore, it's unable to access newly written data in such circumstances and fails to effectively sketch a real dynamic HTAP scenario. Another approach is to randomly instantiate query parameters from the domain of historical and fresh data simultaneously. Nevertheless, it would produce uncertain cardinality sizes, which makes the query computational complexity beyond control. In contrast, *Vodka* achieves predictable changes in cardinalities along with data changes, i.e., a well-controlled computational complexity due to our effective data manipulation and parameter instantiation method.

Next, we compare *Vodka* with *HATtrick* regarding to the consistency model measurement. We compare two representative analytical scenarios, one focusing solely on static historical data and the other on newly modified data [52]. An ideal freshness measurement should return a zero value for the first scenario; while in the second scenario, the returned value should exactly reflect the realistic delay between the commit timestamp of the data version being read in the *A*-engine and that of the latest version in the *T*-engine. Specifically, since PG-SR follows the weak session consistency model that might present different freshness, we thus conduct experiments over PG-SR. Fig. 18 shows the results. Specifically, in the first scenario, we observe that *HATtrick* does not report zero value since it defines the freshness by the difference between the query start time and the commit time of the earliest invisible transaction in the whole system. It does not consider that the version of the static data is already up-to-date. In the second scenario, we follow the evaluation process in § 7.2.1 and use S5 (i.e., e-commerce) as an example for comparison. We notice that *HATtrick* still has a high deviation from the realistic freshness since it does not consider the specific data status being read, while *Vodka* produces the expected result.

Lastly, given that none of the existing work proposes a method for data sharing evaluation, we do not include this comparison.

7.5 System Evaluation

Resource Isolation. We vary throughputs in the *T*-engine and the number of OLAP threads in the *A*-engine to conduct experiments on three systems, which are PG-SR, OceanBase and TiDB. Fig. 20 illustrates the results. For a given number of OLAP threads, we observe that PG-SR has the best effectiveness of isolation and maintains a relatively stable OLAP performance when increasing the OLTP throughputs. This is benefited from its decoupled resource architecture that effectively addresses resource contentions between *T*- and *A*-engines. However, PG-SR's overall performance is relatively low since it lacks optimizations of columnar store and vectorized executions. Moreover, PG-SR struggles with complicated subqueries such as Q18 (> 50s), which further limits its *A*-engine's performance. Meanwhile, we observe that the performance of *A*-engine in OceanBase has a significant decrease when increasing the OLTP throughputs. The reason is that OceanBase adopts a unified resource architecture, and increasing OLTP throughputs would preempt the resources of *A*-engine. Moreover, it uses a single copy of data that is organized by the MVCC mechanism. Then, it constructs longer version chains when it experiences intensive writes, which further decreases OLAP scan efficiency in finding appropriate data versions. Although TiDB also has a unified resource architecture, we observe that the degradation of TiDB's performance is lower than OceanBase. This is because it uses a logical isolation method by binding its TiKV and TiFlash instances to specific CPU cores in each node that helps mitigate resource contentions. Finally, for a given OLTP throughput, we observe that all the systems' OLAP performance does not increase linearly with the scaling of OLAP threads due to multi-thread resource contentions. We notice that OceanBase scales better than other systems since it has optimizations on the inter-thread data exchange and communications [77].



(a) Large Dataset (b) Large Query Volume
(c) Poor Synchronization Ability (d) Long Transaction

Figure 14: Scalability Analysis in Freshness Calculation

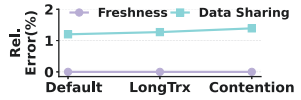


Figure 15: LongTrx&Content.

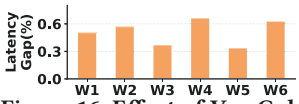


Figure 16: Effect of Ver. Col.

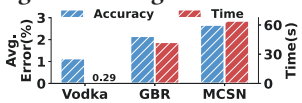


Figure 17: Model Comparison

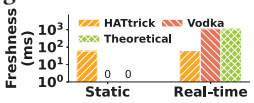
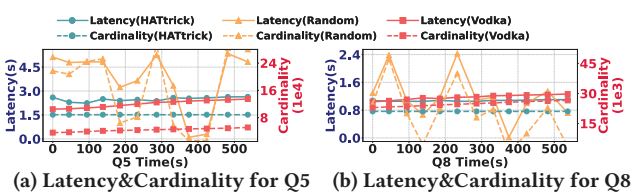


Figure 18: Fresh. Comparison

Consistency Model. We next evaluate the freshness of three systems to compare their ability of real-time analytics. We evaluate each system by executing 100 *ConsistencyCheck* queries (see § 7.1) and report the average freshness results, as summarized in Table 1. Specifically, we observe that OceanBase and TiDB conform to their linear consistency model by providing the freshest data for queries under various throughputs, i.e., $freshness = 0$. We also notice that under a relatively low throughput (≤ 3.5K), PG-SR could also provide the freshest data. However, as the OLTP throughput increases, the freshness of PG-SR significantly decreases. This is because its asynchronous synchronization mechanism could not ensure that the two engines have consistent data versions. In this case, the real-time analytics businesses could not be satisfied since they generally require millisecond-level freshness [61].

Data Sharing. We finally compare the three systems in terms of data sharing efficiency by executing 100 OLAP queries and report the average synchronization latency. Fig. 21 shows the results. We observe that TiDB has the highest latency because it introduces an additional third-party software library named RocksDB [54], which extends the synchronization pathway and increases the logging overhead. We also observe that the synchronization latency in TiDB



(a) Latency&Cardinality for Q5 (b) Latency&Cardinality for Q8

Figure 19: Comparison of Computation Complexity

(a) 1 OLAP thread (b) 5 OLAP threads (c) 10 OLAP threads

Figure 20: The OLAP Performance in Three Systems

and PG-SR increases significantly under higher OLTP throughputs, since they both use a share nothing approach that requires heavy log transfers between the *T*- and *A*-engines. We finally notice that OceanBase has the lowest latency due to its share everything strategy, avoiding expensive data transfers. In conclusion, the share nothing approach incurs significantly higher synchronization latency than the share everything method.

7.6 Experimental Insights

I1: Resource Isolation. The decoupled resource uses physical storage isolation, but it might lead to significant resource idleness in some nodes. This architecture should appropriately pre-allocate resources for the *T*-engine such that its SLA requirements for mission-critical tasks can be satisfied. Moreover, we also need an intelligent strategy to route OLAP queries into the *T*-engine when its resources are low. In contrast, the unified resource employs two strategies. First, the OLTP and OLAP workloads compete for their respective resources (e.g., OceanBase). Although it makes full use of resources, its isolation is unsatisfactory. The second strategy uses logical isolations (e.g., binding cores) to pre-allocate resources for *T*- and *A*-engines (e.g., TiDB), but it might also lead to many idle resources. Therefore, in the unified resource, we need a more effective method that could allocate and schedule resources adaptively based on the real-time OLTP and OLAP resource demands.

I2: Consistency Model. The linear consistency is an optimal choice for high freshness requirements, but it might introduce additional query delays due to waiting for transferring the data. The weak consistency model achieves immediate query responses, but its freshness is not guaranteed. Thus, to satisfy businesses with varying freshness requirements, a flexible approach is to let the architecture support the linear consistency model and allow users to customize queries with different consistency levels by configuring query hints to trade off between data freshness and query latencies, similar to the isolation levels in concurrency control [36].

I3: Data Sharing. Since the share nothing uses asynchronous log transfers between *T*- and *A*-engines for data sharing, its efficiency relies closely on the log writing and replaying strategies. The optimizations, such as parallel log writing and batch log replaying, could be considered. A more preferred approach is to synchronize data on demand, which prioritizes transferring the data that is relevant to queries. The share everything does not require data transfers since the data is organized with MVCC. But it requires efficient version management, such as accelerating the version chain traverses and optimizing garbage collections for stale versions.

I4: Trade-offs. (1) Systems aiming for high performance and high data freshness should first adopt the decoupled resource architecture for efficient resource isolation. We believe that they should further use the linear consistency model to ensure that the latest updates are always merged into the *A*-engine before query executions. However, it inevitably requires a cross-node long data transfer pathway between two engines. (2) Systems aiming for high performance and efficient data sharing should adopt a unified resource architecture. Meanwhile, they should maintain two copies of data in a single node and implement effective logical isolation to mitigate resource contention. As the data copy from the *T*- to *A*-engine in the single node is relatively lightweight, a high data sharing efficiency can be achieved. However, this could not ensure high data freshness as the latest updates might not always be reflected in analytics immediately. (3) Systems aiming for high data freshness and efficient data sharing should adopt a unified resource architecture. They should further use linear consistency and share everything, where two engines share a single copy of data organized with MVCC. This design ensures the highest data freshness and eliminates any data transfers. However, its performance would be negatively affected since hybrid workloads preempt resources intensively.

Table 1: Freshness Result in Three Systems

HTAP Systems	*Vodka* (with different OLTP throughputs)			
	[1K-2K]	3.5K	5K	6.5K
PG-SR	0ms	0ms	2663ms	5208ms
OceanBase	0ms	0ms	0ms	✗
TiDB	0ms	✗	✗	✗

The symbol of ✗ means systems cannot support the specified throughputs.

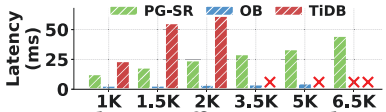


Figure 21: Data Sharing Efficiency in Three Systems

8 RELATED WORK

CH-benCHmark [11] is the first HTAP benchmark that addresses the incompatibility between TPC-C&TPC-H benchmarks. Based on [11], *HTAPBench* [10] introduces a new metric that evaluates the maximum number of OLAP threads a system could support under a specified OLTP throughput. *OLxPBench* [26] is built on TPC-C and includes self-defined OLAP queries. It provides real-time queries within transactions to perform real-time analysis before making a quick decision. Moreover, it offers two domain-specific benchmarks for finance and telecommunication scenarios. However, none of the above studies have designs for freshness and data sharing. *HATtrick* [44] combines SSB with a simplified TPC-C to build an HTAP benchmark. It's the first work to define freshness, i.e., the gap between the query start time and the earliest commit time of *T*-engine's transactions that are invisible to the *A*-engine. However, the definition is not appropriate since users are concerned with the freshness of queried data. *Hybench* [78] designs a new self-defined finance scenario containing new workload types such as analytical transactions. It combines metrics from different dimensions into an integral one. However, none of them enables a fair comparison of resource isolations. Moreover, they lack a reasonable definition of data freshness and do not consider data sharing evaluations.

9 CONCLUSION

In this paper, we introduce *Vodka*, the first comprehensive benchmark that fully evaluates the three key techniques of HTAP systems in a fair way. We conduct extensive experiments on popular HTAP systems, and conclude with novel insights for future researches.

ACKNOWLEDGMENTS

This work is supported by NSFC (92270202, 62202171), the National Key R&D Program of China (2024YFC3308102), and OceanBase, Ant Group Research Fund. Rong Zhang is the corresponding author.

REFERENCES

[1] 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.

[2] Peter Bailis, Aaron Davidson, Alan D. Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. In *Proc. VLDB Endow*, Vol. 7. 181–192.

[3] Altan Birler. 2019. Scalable Reservoir Sampling on Many-Core CPUs. In *SIGMOD*. 1817–1819.

[4] Mokrane Bouzeghoub. 2004. A Framework for Analysis of Data Freshness. In *IQIS*. 59–67.

[5] 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 *Proc. VLDB Endow*, Vol. 11. 1849–1862.

[6] Jacopo Cavazza and Vittorio Murino. 2016. Active Regression with Adaptive Huber Loss. In *CoRR*, Vol. abs/1606.01568.

[7] 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 *Proc. VLDB Endow*, Vol. 15. 3411–3424.

[8] Lixiang Chen, Yuxing Han, Yu Chen, Xing Chen, Chengcheng Yang, and Weining Qian. 2025. AQETuner: Reliable Query-level Configuration Tuning for Analytical Query Engines. In *Proc. VLDB Endow*, Vol. 18. 2709–2721.

[9] Yuxing Chen, Anqun Pan, Hailin Lei, Anda Ye, Shuo Han, Yan Tang, Wei Lu, Yunpeng Chai, Feng Zhang, and Xiaoyong Du. 2024. TDSQL: tencent distributed database system. In *Proc. VLDB Endow*, Vol. 17. 3869–3882.

[10] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2017. Htapbench: Hybrid transactional and analytical processing benchmark. In *ICPE*. 293–304.

[11] 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.

[12] Irving M Copi, Carl Cohen, and Kenneth McMahon. 2016. *Introduction to logic*.

[13] Alain Crolotte and Ahmad Ghazal. 2011. Introducing skew into the TPC-H benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 137–145.

[14] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *SIGMOD*. 969–984.

[15] Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2020. Efficiently approximating selectivity functions using low overhead regression models. In *Proc. VLDB Endow*, Vol. 13. 2215–2228.

[16] 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.

[17] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. In *Proc. VLDB Endow*, Vol. 13. 1162–1175.

[18] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. Hyrise: a main memory hybrid storage engine. In *Proc. VLDB Endow*, Vol. 4. 105–116.

[19] Yuxing Han, Haoyu Wang, Lixiang Chen, Yifeng Dong, Xing Chen, Benquan Yu, Chengcheng Yang, and Weining Qian. 2024. ByteCard: Enhancing ByteDance's Data Warehouse with Learned Cardinality Estimation. In *SIGMOD*. 41–54.

[20] 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 *Proc. VLDB Endow*, Vol. 15. 752–765.

[21] Roman Heinrich, Manisha Luthra, Johannes Wehrstein, Harald Kornmayer, and Carsten Binnig. 2025. How good are learned cost models, really? Insights from query optimization tasks. In *SIGMOD*, Vol. 3. 1–27.

[22] Tianxun Hu, Tianzheng Wang, and Qingqing Zhou. 2022. Online Schema Evolution is (almost) Free for Snapshot Databases. In *Proc. VLDB Endow*, Vol. 16. 140–153.

[23] Zirui Hu, Rong Zhang, Chengcheng Yang, Xuan Zhou, Quanqing Xu, and Chuanhui Yang. 2025. Artemis: A Customizable Workload Generation Toolkit for Benchmarking Cardinality Estimation. In *ICDE*. 4628–4631.

[24] 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 *Proc. VLDB Endow*, Vol. 13. 3072–3084.

[25] Xuhua Huang, Zirui Hu, Siyang Weng, Rong Zhang, Chengcheng Yang, Xuan Zhou, Weining Qian, Chuanhui Yang, and Quanqing Xu. 2025. A Query-Aware Enormous Database Generator For System Performance Evaluation. In *SIGMOD*. 131–134.

[26] 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.

[27] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. 195–206.

[28] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making mvcc systems htap-friendly. In *SIGMOD*. 49–64.

[29] 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.

[30] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. In *Proc. VLDB Endow*, Vol. 13. 2132–2145.

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

[32] 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.

[33] Per-Åke Larson, Adrian Birka, Eric N Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. 2015. Real-time analytical processing with SQL server. In *Proc. VLDB Endow*, Vol. 8. 1740–1751.

[34] 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 *Proc. VLDB Endow*, Vol. 10. 1598–1609.

[35] Guoliang Li and Chao Zhang. 2022. HTAP databases: What is new and what is next. In *SIGMOD*. 2483–2488.

[36] Keqiang Li, Siyang Weng, Peiyuan Liu, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, Jianghang Lou, Gui Huang, Weining Qian, et al. 2023. Leopard: A black-box approach for efficiently verifying various isolation levels. In *ICDE*. IEEE, 722–735.

[37] Keqiang Li, Siyang Weng, Lyu Ni, Chengcheng Yang, Rong Zhang, Xuan Zhou, and Aoying Zhou. 2024. DBStorm: Generating various effective workloads for testing isolation levels. In *ISSTA*. 755–767.

[38] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: Generating Enormous Query-Aware Test Databases. In *ATC*. 575–586.

[39] Yishuai Li, Yunfeng Zhu, Chao Shi, Guanhua Zhang, Jianzhong Wang, and Xiaolu Zhang. 2024. Timestamp as a Service, not an Oracle. In *Proc. VLDB Endow*, Vol. 17. 994–1006.

[40] 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.

[41] 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.

[42] 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.

[43] Sergey Melnik, Atul Adya, and Philip A Bernstein. 2008. Compiling mappings to bridge applications and databases. In *TODS*, Vol. 33. 1–50.

[44] 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.

[45] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedon Chen. 2007. The star schema benchmark (SSB). In *Pat*, Vol. 200. 50.

[46] Oracle. 2024. Real-Time Access to Real-Time Information. https://www.hunkler.de/files/downloads/oracle_wp_golden-gate-12c-real-t.pdf.

[47] Zhicheng Pan, Yuanjia Zhang, Chengcheng Yang, Ahmad Ghazal, Rong Zhang, Huiqi Hui, Xiaoju Wu, Yu Dong, and Xuan Zhou. 2025. Hyper: Hybrid Physical Design Advisor with Multi-agent Reinforcement Learning. In *ICDE*. IEEE, 1565–1578.

[48] 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.

[49] PostgreSQL. 2024. PostgreSQL. https://www.postgresql.org/.

[50] PostgreSQL. 2024. Streaming Replication Protocol. https://www.postgresql.org/docs/current/protocolreplication.html.

[51] Luyi Qu, Qingshuai Wang, Ting Chen, Keqiang Li, Rong Zhang, Xuan Zhou, Quanqing Xu, Zhifeng Yang, Chuanhui Yang, Weining Qian, et al. 2022. Are current benchmarks adequate to evaluate distributed transactional databases?. In *BenchCouncil*, Vol. 2. 100031.

[52] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In *SIGMOD*. 2043–2054.

[53] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *SIGMOD*. 1583–1598.

[54] RocksDB. 2024. https://rocksdb.org.
[55] Stuart J Russell and Peter Norvig. 2010. *Artificial intelligence a modern approach*.
[56] Tobias Schmidt, Dominik Durner, Viktor Leis, and Thomas Neumann. 2024. Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP. In *Proc. VLDB Endow*, Vol. 17. 3290–3303.
[57] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting high availability mechanism to tame hybrid transaction/analytical processing. In *OSDI*. 219–238.
[58] Dong Keun Shin and Arnold Charles Meltzer. 1994. A new join algorithm. In *SIGMOD*, Vol. 23. 13–20.
[59] Niharika Singh and Ashutosh Kumar Singh. 2018. Data privacy protection mechanisms in cloud. In *Data Science and Engineering*, Vol. 3. 24–39.
[60] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance characterization of HTAP workloads. In *ICDE*. 1829–1834.
[61] Haoze Song, Wenchao Zhou, Heming Cui, Xiang Peng, and Feifei Li. 2024. A survey on hybrid transactional and analytical processing. In *VLDBJ*. 1–31.
[62] Hennie J Steenhagen, Peter MG Apers, Henk M Blanken, and Rolf A de By. 1994. From nested-loop to join queries in oodb. In *Proc. VLDB Endow*. 618–629.
[63] Wei Chit Tan. 2018. Efficient Query Reverse Engineering for Joins and OLAP-Style Aggregations. In *APWeb/WAIM (2)*. 53–62.
[64] TiDB. 2024. Hybrid Deployment Topology. https://docs.pingcap.com/tidb/stable/hybrid-deployment-topology.
[65] TPC. 2024. TPC-DS Benchmark. http://www.tpc.org/tpcds/.
[66] TPC. 2024. TPC-H Benchmark. http://www.tpc.org/tpch/.
[67] 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.
[68] 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 *Proc. VLDB Endow*, Vol. 15. 1991–2004.
[69] Vodka. 2025. Technical Report And Source Codes Of Vodka. https://github.com/DBHammer/Vodka-2025.
[70] 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.
[71] 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.
[72] Wikipedia. 2024. Bayes' theorem. https://en.wikipedia.org/wiki/Bayes%27_theorem.
[73] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. In *Proc. VLDB Endow*, Vol. 10. 781–792.
[74] Quanqing Xu, Chuanhui Yang, and Aoying Zhou. 2024. Native Distributed Databases: Problems, Challenges and Opportunities. In *Proc. VLDB Endow*, Vol. 17. 4217–4220.
[75] Hiroyuki Yamada, Kazuo Goda, and Masaru Kitsuregawa. 2023. Nested loops revisited again. In *ICDE*. IEEE, 3708–3717.
[76] 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 *Proc. VLDB Endow*, Vol. 13. 3313–3325.
[77] 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 *Proc. VLDB Endow*, Vol. 16. 3728–3740.
[78] Chao Zhang, Guoliang Li, and Tao Lv. 2024. HyBench: A New Benchmark for HTAP Databases. In *Proc. VLDB Endow*, Vol. 17. 939–951.
[79] 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.
[80] Junpeng Zhu, Zhiwei Ye, Peng Cai, Donghui Wang, Fengyan Zhang, Dunbo Cai, and Ling Qian. 2024. Log Replaying for Real-Time HTAP: An Adaptive Epoch-Based Two-Stage Framework. In *ICDE*. 2096–2108.