Vodka: Fairly Benchmarking HTAP Systems on Demand [Experiment, Analysis & Benchmark]

Zirui Hu
ECNU
zrhu@stu.ecnu.edu.cn

Qingshuai Wang
ECNU
qswang@stu.ecnu.edu.cn

Yao Luo
ECNU
yaoluo@stu.ecnu.edu.cn

Rong Yu
ECNU
ryu@stu.ecnu.edu.cn

Jinkai Xu
ECNU
jinkaixu@stu.ecnu.edu.cn

Rong Zhang
ECNU
rzhang@dase.ecnu.edu.cn

Xuan Zhou
ECNU
xzhou@dase.ecnu.edu.cn

Aoying Zhou
ECNU
ayzhou@dase.ecnu.edu.cn

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

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

ABSTRACT

For real-time analysis on latest modifications, hybrid transaction/-analytical processing (HTAP) systems have recently gained popularity. Unfortunately, we still lack a benchmark to comprehensively compare the two distinct key techniques, i.e., *resource isolation* and *data sharing*, in different HTAP systems. In this paper, we argue that to provide repeatability and fairness besides real-time queries, semantics consistent HTAP scenario and unified metrics is essential for benchmarking HTAP systems. However most of the state-of-the-art benchmarks can not satisfy all these requirements. So this paper proposes $Vodka$, a new benchmark framework for HTAP systems. $Vodka$ contributes to: 1) formulate the query cardinality control and keep data access distribution consistency under a dynamic data change for benchmark repeatability; 2) model the data size and query performance for a consistent comparison; 3) propose a query-oriented freshness definition and design a workload interaction pattern for simulating various data sharing pressure in HTAP scenarios; 4) define new metrics for fair comparisons under different data sharing models. Finally, we conduct extensive experiments on three representative HTAP systems to justify the designs of $Vodka$ and provide insights for implementing HTAP systems. Its source code is available [54].

1 INTRODUCTION

To satisfy the increasing demand for real-time analytics, e.g., online exception detection [44] or accurate recommendation [47], after a long separation of OLTP-engine (T-engine) and OLAP-engine (A-engine), we are starting to execute hybrid workloads in a class of new engines, which are called Hybrid Transactional and Analytical Processing (HTAP) systems [13].

New Technical Demands For HTAP. Traditionally, T-engine typically operates on a limited number of tuples, and use index to accelerate random IO access [62]. The latency-sensitive feature drives T-engine to enhance update and read efficiency by row-based storage, focusing more on optimizing cache hit rates and network latency. In contrast, A-engine generally handles workloads for large-scale sequential IO accesses. The design goal of improving overall query throughput drives A-engine to store data in a column-based storage, consuming substantial CPU and memory resources to parallelize data reading through single-instruction multiple-data (SIMD) instructions [26]. It is obvious that T- and A-engines are not consistent in their implementations. To alleviate the heterogeneity and guarantee performance of hybrid workloads, a massive flurry of giant database vendors participate in constructing HTAP systems [5, 7, 12, 17, 19, 22, 25, 59, 61]. Typically, we divide the implementation architectures into two types called unified storage and decoupled storage based on the way of sharing storages [45, 53]. In unified storage HTAP systems, e.g., Oracle [22], they tend to execute hybrid workloads in a single node with no extra data transferring. To mitigate interference from each other, they try to retrofit traditional technologies like multiversion concurrency control (MVCC) and snapshotting mechanisms, with the priority of OLTP in row-based storage. Alternatively, they maintain dedicated in-memory column storage for OLAP and fetch logs of OLTP to update the column storage. The decoupled storage is also taken by many mainstream HTAP systems, e.g., F1 Lightning [59], which separate T- and A-engines on different nodes with the purpose of a resource isolation. Then we can import fresh data from T-engine asynchronously into A-engine, which generally achieves a relatively high performance at the cost of data freshness. In summary, to provide HTAP services, different architectures or optimizations are designed to trade off $performance$ against $freshness$.

The Design Philosophy of HTAP Benchmarks. To assess the pros and cons of HTAP systems, benchmarking has always been a fair way for comparison. Traditionally, there has been extensive research on benchmarking OLTP/OLAP systems, while HTAP benchmark can not be a simple stitch of OLTP/OLAP benchmark [18]. Besides OLTP and OLAP services provided by HTAP systems, the main distinct characteristic of HTAP systems is the fresh data consumption of A-engine, which means the real-time data access overlapping between OLTP and OLAP workloads [18]. It implies that 1) the **computational complexity** of an OLAP workload is highly affected by the produced data from T-engine; 2) the data access interleaving size between OLAP workloads and the latest modification from OLTP workloads makes **synchronization and data reorganization cost** dynamically change. Though existing benchmarks have considered some HTAP business features, such as introducing real-time queries for data from the runtime transactions, the more

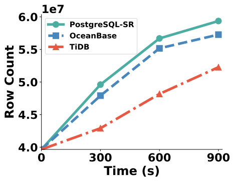
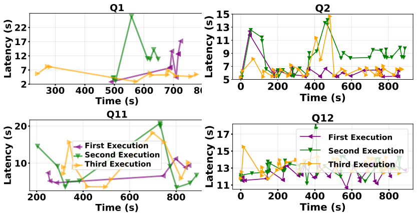


Figure 1: Data Increase in Three HTAP Systems



(a) CH-benchmark

(b) HATtrick

Figure 2: Stability of Query Latency

critical issues like the dynamic changes in OLAP computational complexity caused by runtime OLTP modifications, as well as the data synchronization and reorganization costs arising from different scales of fresh data access, have not been adequately considered. To facilitate a fair and indepth comparison, it is essential to quantify or control these dynamics for HTAP workloads. We summarize the main challenges as followings:

Distinct System Processing Capacity *vs*. **Consistent Workload Complexity Requirement.** T-engines of different HTAP systems can expose distinct OLTP processing capacities and then generate different new data size at the same time interval. As shown in Fig. 1, we monitor data increasing (row count) of tables *Order* and *Orderline* in TPC-C under the peak OLTP performance of the three HTAP systems (configurations in § 6) and find that the data increasing varies obviously. It means the same query submitted at the same time point to different A-engines will meet totally different computational complexity. Moreover, distinct capabilities of A-engines can lead to differences in execution latencies for the same query. Then even if T-engines present the same TPS, the same queries on different A-engines will access different sizes of (synchronized) data. Consequently, the cardinalities of inputs for OLAP query operators become uncertain, resulting in fluctuations in query latency across multiple rounds of tests. As shown in Fig. 2, we run the widely adopted CH-benchmark [11] and the state-of-the-art HATtrick [31] on the classic HTAP system PostgreSQL-SR [41] by configuring 8 OLTP and 8 OLAP threads. We expose the performance of two queries in each benchmark. It is obvious that all performance fluctuates badly, i.e., bad benchmarking repeatability. Thus, the distinct capability of T- and A-engines makes it tough to align the data to have the same computational complexity.

Nondeterministic Data Access Interleaving *vs*. **Deterministic Interaction Pattern Orchestrating Requirement.** HTAP systems require the newly generated data of T-engine to be accessed by A-engine within a freshness threshold [47]. The degree of freshness and the volume of freshness data accessed by A-engine have an obvious impact on the performance of HTAP systems, for it may produce variant synchronization pressures and resource contentions. Therefore, quantifying the synchronization cost or orchestrating this nondeterministic overlapping/interleaving relationship and provide a deterministic and controllable interaction pattern between A- and T- engines for various query scenarios is the core challenge in evaluating the real-time processing capability.

Multi-dimensional Metrics *vs*. **Unified Comparability Requirement.** Different applications have various freshness requirements, while high freshness tends to sacrifice performance [47], which inspires the existence of various architectures to provide diverse data freshness for analysis and achieve a performance-freshness trade-off. So the reported excellent query performance can be misleading due to its weak requirement for synchronized fresh data. For a fair comparison, we need to define a unified new metric by integrating performance with freshness to avoid the bias. However, existing metrics for HTAP system performance, like query latency and freshness, are not of the same scale, so they can not be simply integrated for a comprehensive capability comparison among HTAP systems.

Table 1: Resource Isolation Techniques

Resource Isolation				
Isolation Degree		**Logical**	**Storage**	
			Memory	**Disk**
Type	**CPU**	Virtualization [29]	Physical Isolation	Physical Isolation
	Memory	Delta-Versioning [12] MVCC [21]; CoW [19]		
	Disk	/	/	
Representative HTAP Systems		Greenplum [29]; Oracle [22] OceanBase [60]; Hyper [19]	PolarDB [5] Aurora [52]	PostgreSQL-SR [42];Vegito [47] TiDB [17]; BatchDB [30]

To deal with the challenges in benchmarking HTAP systems, we propose a new benchmarking framework *Vodka* with its contributions summarized as followings. (1) We are the first work to formalize query parameter regulation with the purpose of controlling query computational complexity under dynamic data changes for the benchmarking repeatability (in §3). A Huber Loss-based log-linear model is then designed to sketch the relationship between query latency and the data size for a consistent comparison. (2) We provide a formal definition to workload interaction pattern for simulating data sharing pressure accurately, propose a new query-oriented freshness definition and design a unified metric to consolidate both freshness and query latency for exposing the HTAP ability fairly (in §4). (3) We provide the implementation details for constructing a semantics consistent HTAP application in data/schema, and workloads (in § 5). (4) We conduct extensive experiments to justify the designs in *Vodka* and provide insights for implementation HTAP systems (in § 6).

2 BACKGROUND

Due to different workload characteristics of T-engine and A-engine, executing hybrid workloads in a single system will naturally cause resource contention and performance degradation [11]. As a result, to guarantee performance under data sharing requirement, a primary task for an HTAP system is to provide an effective *resource isolation* technique. If A-engine can read the latest production data from T-engine, it provides the highest freshness, which usually imposes a severe data organization or communication demand to T-engine [30]. Therefore, to guarantee the performance of each engine, the other task for an HTAP system is to provide an efficient *data sharing* model for freshness data [53]. So, compared to the single-type systems, HTAP systems devote the effort to **achieving efficient data sharing under effective resource isolation**.

2.1 Resource Isolation and Data Sharing

2.1.1 Resource Isolation. It refers to scheduling and separating the hardware resources to mitigate resource contention between two types of engines. We divide the classic implementation techniques into two categories, i.e., *logical isolation* and *storage isolation*.

Table 2: Comparison of State-of-the-art HTAP Benchmarks (T: Support, F: Not Support, P: Partial Support)

Benchmark			CH-benchmark	HTAP-Bench	Swarm64	OLxPBench	HATtrick	**Vodka**
Application Scenario		Semantically-Consistent Scenario	F	F	F	T	T	T
		Representative OLAP&OLTP Workloads	T	T	T	P	P	T
Key Techniques	**Resource Isolation**	Fair OLTP/OLAP Performance Measurement	F	F	F	F	F	T
		Precise Interference Degree Measurement	P	T	T	P	P	T
	Data Sharing	Freshness Calculation Method	F	F	F	F	P	T
		Data Sharing Pressure Control	F	F	F	F	F	T
Metrics		Unified Comparability	F	F	F	F	F	T

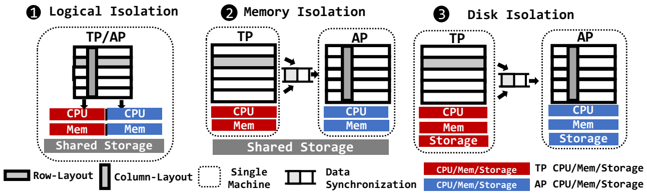


Figure 3: Overview of Resource Isolation

For Logical Isolation, both *T*- and *A*-engines share all resources of the same machine, and we use logical methods to schedule resources [45]. For CPU, it can use virtualization techniques to split CPU resources into groups *w.r.t* hybrid workloads [29, 60]. Or else, we can bind CPUs to different NUMA nodes to mitigate contention [30, 49]. For memory, according to the granularity of data versions, it can be further classified into three types. The first one is to maintain two data replicas for serving OLAP reads and OLTP writes respectively [12, 14, 22]. The second one is based on the *copy-on-write (CoW)* mechanism [19], which reads by creating the latest snapshot of data without blocking OLTP updates. Both of these two types have to fetch updates from *T*-engines with expensive communication and re-organization cost. The last option is to share the *multiversions* of tuples (MVCC) [58] created by *T*-engine to *A*-engine [21]. Though it avoids the expensive cost of data copy or synchronization, *A*-engine may meet an expensive version traverse cost to identify the consistent versions of tuples for a query [21, 47].

For Storage Isolation, it has *memory isolation* and *disk isolation*. In memory isolation, both *T*- and *A*-engines share storage resources in the same machine, but they use isolated CPU and memory resources of different machines [5, 52, 52].But OLAP clients may have to solve the problem of data inconsistency between the memory and storage layers. In disk isolation, *T*- and *A*-engines are put separately on different machines, i.e., an absolute hardware isolation. *T*-engine synchronizes its logs or data to *A*-engine for fresh data accessing [17, 30, 42, 47]. For the fresh data, *A*-engine will ask *T*-engine to transfer the latest updates, causing a high network cost and increasing query latency.

2.1.2 Data Sharing. It refers to the method of sharing the produced data from *T*-engine to *A*-engine [48]. There may exist version gaps or time delays between *T*- and *A*-engines for the same data. The gap follows the definitions of **data sharing models** (*DSM*s), which guarantees different freshness degrees, including *linear consistency*, *sequential consistency*, and *session consistency*. We put the formal definition of *DSM*s in our technical report [54].

Linear Consistency refers to that the data accessed on *A*-engine is exactly the same as the latest one on *T*-engine , which is widely adopted by SAP HANA [12], Hyrise [14], TiDB [17]. In practice, it's usually implemented on a unified storage database [12, 14, 22, 60]. While for a database with decoupled storage, it tends to transfer OLTP logs asynchronously and performs a check to confirm whether it is the latest version for OLAP queries [17]. Sequential Consistency allows *A*-engine to access previous snapshot versions, but all *A*-engines should read the consistent same version of the snapshots at the same time-point. It is adopted by PostgreSQL-SR [42], BatchDB [30] and Vegito [47]. In practice, most of them are used for HTAP systems with decoupled storage, which synchronizes OLTP data in epochs (batches). Session Consistency only requires each client to see data following the strict version generation order from *T*-engine, supported by PolarDB [5], Aurora[52] and Hyper[19]. In practice, it is often implemented by maintaining individual data snapshots in different caches with shared storage; it can be also employed by the stand-alone HTAP system by creating a series of session snapshots.

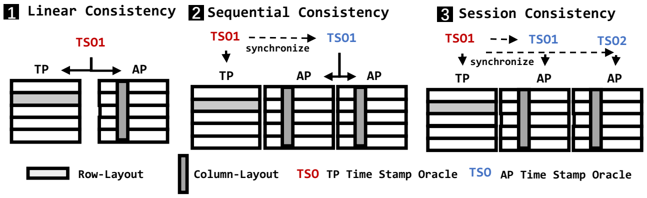


Figure 4: Data Sharing Models in HTAP Systems

2.2 Related Work

Compared to traditional single-type database systems, HTAP systems have their distinct characteristics of providing both OLAP and OLTP services 1) under **a unified application scenario**; 2) by **resource isolation** for stable service; 3) by **data sharing** for fresh data consumption, and with the tailored 4) **metrics** for these new characteristics. According to these requirements, we identify the limitations of current mainstream HTAP benchmarks in Table 2.

First, HTAP systems are deployed to sever applications with hybrid workloads. Under the *unified application scenario*, it is imperative to guarantee semantic consistency for table schema, data distribution as well as workload execution logic. Most of the benchmarks stitch the existing benchmarks. For example, CH-benCHmark [11], HTAPBench [10], and Swarm64 [40] are based on TPC-C and TPC-H considering little semantics consistency. The deficiency is first mentioned by OLxPBench [18]. However, OLxPBench focuses only on the consistency of data schema, based on which it constructs simple OLAP queries to access a consistent schema adapted from TPC-C. Lacks of complex queries/ predicates make the performance evaluation incomprehensive. Likewise, HATtrick [31] constructs simple transactions based on SSB. The simplified OLTP and OLAP workloads make it not a representative HTAP application scenario.

Second, HTAP systems use *resource isolation* technique to mitigate resource contention between *T*- and *A*-engines with the purpose of reducing performance interference and guaranteeing stable OLTP/OLAP performance [45]. Then benchmarking resource isolation should provide a fair measurement of OLTP/ OLAP performance under a precise control of the mutual interference. As far as we know, no-existing work takes the OLTP throughput together with the data sharing models among systems into consideration, which will cause OLAP queries meet naturally unequal data size. To compare the interference between T- and A-engines, it usually uses the observation of OLTP (resp. OLAP) performance degradation degree when adding more OLAP (resp. OLTP) threads. It is adopted by HATtrick and OLxPBench. But different engines have different processing ability with the same client threads, that is the data generation speed from T-engine and data consumption or fresh data synchronization requirement from A-engine may differ from each other. Usually the benchmark results can have obvious bias from different rounds of runs for different or even the same HTAP systems because of the uncontrollable data size involved in computation. It can not achieve a fair comparison.

Third, no matter how isolation is, T-engine should share data with A-engine for various freshness requirements following its *data sharing model* [30]. Meanwhile, users care about whether the data in the access range of their queries is fresh enough [4, 37], instead of the global freshness state of the whole system. To the best of our knowledge, current benchmarks all fail to adequately cover these requirements. Though CH-benCHmark has a definition of freshness, it provides no measurement method. HATtrick focuses only on the global freshness state of the system. Meanwhile, none of them can quantify the data sharing pressure derived from the interaction in hybrid workloads.

Finally, we usually classify HTAP system architectures based on the sharing of storage into *unified* and *decoupled* storage [45]. Existing benchmarks do not distinguish data sharing models. Then it leads to a deficiency in exploring the comprehensive relationship between freshness and performance. For instance, low query latency for real-time queries may stem from incomplete synchronization. The trade-off between performance and freshness requires us to focus more on the comprehensive relationship between them.

After a thorough study of the benchmark ability in Table. 2, we observe that none of the existing work can perfectly satisfy benchmarking requirements of HTAP systems. It inspires us to present our benchmark framework $Vodka$, which is designed to bridge these gaps, expose the merits of different HTAP systems, and achieve a repeatable and fair comparison among HTAP systems.

3 COMPUTATIONAL COMPLEXITY CONTROL

Since T-engine has a strict requirement of low latency, it is common to guarantee the stable throughput of T-engine when benchmarking HTAP systems [22, 30, 47]. We follow the same way, and then all HTAP systems are assumed to have the same increase of new data after the same period of time. With this precondition, benchmarking HTAP systems becomes to guarantee the repeatability and fairness evaluation to A-engines as declared in traditional OLAP benchmarks, that is different rounds of benchmarking expose almost the same performance. It is achieved by keeping a stable computational complexity of operators, which is closely related to the operator cardinality and data access distribution [28, 46]. However, the dynamic data changes derived from OLTP modifications with different consistency requirements introduce uncertainty to the output cardinality of operators and data access distribution of queries, resulting in unpredictable query complexities or unstable query latencies during benchmarking. It is imperative for us to quantify the cardinalities of SPJA operators, i.e., Selection, Join, Projection and Aggregation, for a given data size (in § 3.1) and control a stable access distribution (in § 3.2). It then facilitates the modeling of query latency under different data sizes in § 3.3 to achieve a fair comparison among HTAP systems with different processing ability.

3.1 Query Cardinality Control

Specifically, for a query, the cardinality of each operator depends on the size and distribution of the input data involved by its query parameters. To quantify the output cardinality, a dynamic query parameter regulation according to the changing of data is imperative. Considering a query q, $Rel(q)$ denotes the relations involved in q. The logical execution plan of q can be represented by a query tree. Each leaf node is a relation R_i ($R_i \in Rel(q)$); $\sigma_{P_{S_i}}$ is a selection with a predicate P_{S_i} on relation R_i; $R_i.A_k$ is column A_k of relation R_i. $R_i \bowtie_{P_J} R_j$ denotes a join operation between two relations R_i and R_j with a join predicate P_J. The projection and aggregation operator are denoted as Π and $\mathcal{G}$. Since the projection and aggregation are often at the top node of a query tree, we prioritize discuss the cardinality control of join and selection operators.

Note that many state-of-the-art OLAP benchmarks [35, 50, 51] tend to conduct queries by instantiating various parameters in the query templates on the static datasets but require to maintain a strict selectivity stability of selection predicates from the same query template. That is the parameterization option should guarantee the equivalence in query complexity, (a.k.a. cardinality size) under a stable data distribution. To vary the parameters of selection predicates while keeping the cardinality of all other operators stable, OLAP benchmarks tend to follow two stipulations, which are **Inter-column Independency of Selection** (R_i and R_j) on their predicates (P_{S_i} and P_{S_j}), represented by $\sigma_{P_{S_i}} \perp \sigma_{P_{S_j}}$ in Eqn. 1 and **Independency between Join and Selection**, represented by $\sigma_{P_{S_i}} \perp\bowtie_{P_J} (R_i, R_j)$ and $\sigma_{P_{S_j}} \perp\bowtie_{P_J} (R_i, R_j)$ in Eqn. 2.

$$Pr(\sigma_{P_{S_i}}|\sigma_{P_{S_j}})=Pr(\sigma_{P_{S_i}}), Pr(\sigma_{P_{S_j}}|\sigma_{P_{S_i}})=Pr(\sigma_{P_{S_j}}) \tag{1}$$

$$\frac{|\sigma_{P_{S_i}}|}{|R_i|}=\frac{|\sigma_{P_{S_i}}(R_i \bowtie_{P_J} R_j)|}{|R_i \bowtie_{P_J} R_j|}, \frac{|\sigma_{P_{S_j}}|}{|R_j|}=\frac{|\sigma_{P_{S_j}}(R_i \bowtie_{P_J} R_j)|}{|R_i \bowtie_{P_J} R_j|} \tag{2}$$

Inter-column Independency of Selection (σ-Indep) means the filter selectivity $Pr(\sigma_{P_S})$ in one relation is generally not influenced by the selectivity of the column from the other relation. For example, Q10 in TPC-H has a join between *Orders* and *LineItem* relations, which selects orders within a date range of three months with the predicate as *o_orderdate >= date '?' and o_orderdate < date '?' + interval '3' month* to join with *LineItem* that have been rejected by customers with the predicate as *l_returnflag = 'R'*. The *date* in *Orders* and the *return flag* in *LineItem* are mutually independent.

Independency between Join and Selection ($\bowtie$-Indep) means that under the same filter selectivity, the value of the parameter in the selection predicate will not affect the join cardinality size. For example, Q5 in TPC-H has the selection predicate as *r_name = '?' and o_orderdate >= date '?' and o_orderdate < date '?' + interval '1' year*, which involves *region name* in *Region* and *order date* in *Orders*. In this context, various parameter options can be adopted for the same selectivity, but always ensure the same join cardinality.

Since selection and join operators satisfy the *Commutation Law* [36], the **SEL**ectivity (Sel) of the output from the join ($\bowtie$) and selection (σ) operators can be generalized as the probability (Pr) of generating results from R_i and R_j satisfying selection and join predicates, i.e., P_S and P_J, and it can be further viewed as the probability of the situation where both the selection and join predicates are satisfied regardless of the concrete execution order of them as in Eqn. 3. According to Bayes theorem [56], Eqn. 3 can be equally represented by Eqn. 4. Based on the σ-Indep and $\bowtie$-Indep, we finally deduce Eqn. 5 and Eqn. 6.

$$Sel(\bowtie, \sigma) = Pr(\sigma_{P_{S_i}} \bowtie_{P_J} \sigma_{P_{S_j}}) = Pr(\sigma_{P_{S_i}}, \sigma_{P_{S_j}}, \bowtie_{P_J}) \tag{3}$$

$$=Pr(\sigma_{P_{S_i}}, \sigma_{P_{S_j}} \mid \bowtie_{P_J}) * Pr(\bowtie_{P_J}) \tag{4}$$

$$=[Pr(\sigma_{P_{S_i}} \mid \bowtie_{P_J}) * Pr(\sigma_{P_{S_j}} \mid \bowtie_{P_J})] * Pr(\bowtie_{P_J}) \quad //\sigma_{P_{S_i}} \perp \sigma_{P_{S_j}} \tag{5}$$

$$=Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * Pr(\bowtie_{P_J}) \quad //\sigma_{P_{S_i}} \perp\bowtie_{P_J}, \sigma_{P_{S_j}} \perp\bowtie_{P_J} \tag{6}$$

Based on Sel, the **CARD**inality ($Card$) from the join and selection can be generally calculated by Eqn. 7, with $|R_{i/j}|$ as the table size.

$$\begin{aligned} Card(\bowtie, \sigma) &= Sel(\bowtie, \sigma) * |R_i| * |R_j| = Pr(\sigma_{P_{S_i}} \bowtie_{P_J} \sigma_{P_{S_j}}) * |R_i| * |R_j| \\ &= Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * (Pr(\bowtie_{P_J}) * |R_i| * |R_j|) \\ &= Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * |R_i \bowtie_{P_J} R_j| \end{aligned} \tag{7}$$

Under the two independence assumptions, the final cardinality is decided by quantifying the join cardinality from the join predicate in $|R_i \bowtie_{P_J} R_j|$, and the selection selectivity from selection predicates, i.e., $Pr(\sigma_{P_{S_i}})$ and $Pr(\sigma_{P_{S_j}})$. Even though we can ensure the aforementioned mutual independencies, data is evolving due to the modifications from T-engine in an HTAP system, which becomes $R'_{i/j}$ by increasing $\Delta_{i/j}$ for $R_{i/j}$. The selection predicates $P_{S_{i/j}}$ may filter out data from both $\Delta_{i/j}$ and the original relations $R_{i/j}$, and then the cardinality from the join and selection operators is formulated by Eqn. 8.

$$\begin{aligned} Card(\bowtie, \sigma) = Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * |R'_i \bowtie_{P_J} R'_j| \\ s.t. \quad R'_i = R_i \cup \Delta\mathrm{R_i} \text{ and } R'_j = R_j \cup \Delta\mathrm{R_j} \end{aligned} \tag{8}$$

The output cardinality in Eqn. 8 is related to both the selectivity of selection predicates $Pr(\sigma_{P_{S_{i/j}}})$ and selectivity of join predicate $Pr(\bowtie_{P_J} (R'_i, R'_j))$. We will then give a thorough discussion of the way to guarantee the stable selectivity of the selection predicates (in §3.1.1), and the stable cardinality output from the join operator (in §3.1.2), after which we discuss the output control for projection and aggregation operator (in §3.1.3).

3.1.1 Selection Selectivity Control. We first analyze the selectivity of selection predicates, e.g., $Pr(\sigma_{P_S})$, and seek to make it predictable even in the dynamic scenarios. The selectivity α ($0 \leq \alpha \leq 1$) is calculated beforehand. Take a selection predicate '$l_i \leq A_i \leq r_i$' as an example with l_i and r_i as the boundary parameters of the filter on column A_i. Varying the pairs of parameters (l_i, r_i) can generate various α. To avoid high traversal cost in a large-scale relation, we use reservoir sampling [1, 3, 57] to tame this problem, which provides a scalable way to sample the dynamic data in a single pass and generates a fixed size of sample results (in slots). Given an error bound δ and a confidence level $\hat{\rho}$, Hoeffding's Inequality can be employed to determine the required size N of the reservoir for sampling [15], where $N = \frac{ln2-ln(1-\hat{\rho})}{2\delta^2}$ with $\delta, \hat{\rho} \in [0, 1]$.

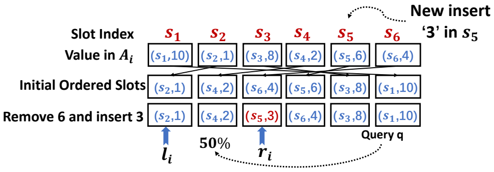


Figure 5: An Example of Reservoir Sampling

For example, we have a reservoir with 6 slots (from s_1 to s_6 as in Fig. 5) sampled from column A_i. The value in each entry of the slot is also paired with its slot index. Specifically, (s_1,10) means value 10 is sampled into slot s_1. To facilitate a quick slot searching and slot updating, we order slots on the sampled values, which has the slot updating complexity of $O(logN)$. In Fig. 5, the example initial ordered slots are $s_2, s_4, s_6, s_5, s_3, s_1$. If we sample a new value 3 to update slot s_5, the slots are reordered accordingly with time complexity of $O(N)$.

We now discuss the method of parameter instantiation for a selection predicate P_{S_i} given a selectivity α based on the sampled reservoir. Note that P_{S_i} can be represented by the conjunctive normal form (CNF) of $clause_x$, which is a disjunctive normal form (DNF) of $literal_{xy}$ as in Eqn. 9.

$$P_{S_i} = \wedge_{x=1}^{n} clause_x \quad s.t. \quad clause_x = \vee_{y=1}^{m} literal_{xy}. \tag{9}$$

For instance, a selection predicate template P_{S_i}=$\underline{(A_1{<}r_1 \vee A_4{>}l_2)}\wedge \underline{(A_2{+}A_3{<}r_3)}$ has $clause_1$:$A_1{<}r_1 \vee A_4{>}l_2$ and $clause_2$:$A_2{+}A_3{<}r_3$. The $\underline{literal_{xy}}$ in $clause_x$ can be a unary or an arithmetic predicate. Specifically, a unary predicate usually follows the form of $A_i \bullet val$ with $\bullet \in \{=, \neq, <, >, \leq, \geq, (not)\ in, (not)\ like, between\ and\}$, e.g., $A_1{<}r_1$. An arithmetic predicate can operate on multiple attributes $A_i, ..., A_k$ via a function $g()$, which generally can be represented as $g(A_i, ..., A_k) \circ val$, $\circ \in \{<, >, \leq, \geq\}$, e.g., $A_2{+}A_3 < r_3$. Based on the sampled reservoir, to instantiate parameters, i.e., r_1, l_2, r_3, related to A_1, A_2, A_3 and A_4 for the expected selectivity α in P_{S_i} is then classified into the following four cases.

Case 1. n=1 & m=1: The simplest predicate is P_{S_i}=$literal$ in Eqn. 9. The predicate is generalized as P_{S_i}=$l_k \leq A_k \leq r_k$. Specifically, the operator of '=' is a special case with $l_k = r_k$ while the operator of '≠' is the reverse of '=' with the selectivity of 1-α for operator '='. To instantiate l_k and r_k for the target selectivity α, the left boundary l_k can be picked from the first $N * (1 - \alpha)$ slots, and then r_k is determined by $l_k + N * \alpha$. For the example in Fig. 5 with α=50%, if we select l_k=1 at the first entry, we can directly locate the half place of the reservoir with $O(1)$ complexity, i.e., l_k=1 and r_k=3, which guarantees the 50% selectivity of P_{S_i}.

Case 2. n>1 & m=1: The case is that a predicate has n $clause$s, each of which has only one $literal$, i.e., P_{S_i} =$\wedge_{k=1}^{n} l_k \leq A_k \leq r_k$ in Eqn. 9. If columns $A_{k_1}, \cdots, A_{k_v}$ are independent, we maintain the reservoir for each column A_{k_j} ($k_1 \leq k_j \leq k_v \leq n$), so that their selection boundaries can be decided independently according to the selectivity of α_{k_j}. If $A_{k_1}, \cdots, A_{k_v}$ are inter-dependent, the permutation space of these columns is taken together to construct the ordered reservoir. The entry of each slot is $(s_i, A_{k_1}, \cdots, A_{k_v})$, with s_i as the slot index, and sorted in order from left (A_{k_1}) to right (A_{k_v}). Then we instantiate the boundaries related to the v dependent columns based on the v-dimensional reservoir slots. When the literal

Table 3: Boundary Values in Selection Predicates ($\epsilon > 0$)

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

has an arithmetic computation as P_{S_i}=$g(A_{k_1}, \cdots, A_{k_v}) \leq r$, it builds a reservoir based on the results of $g()$ function on the v columns, and the entry is $(si, g(A_{k_1}, \cdots, A_{k_v}))$, which is used to instantiate the parameters related to $g()$.

Case 3. n=1 & m>1: The case is that a predicate has one *clause* with more than one *literal*s, i.e., $P_{S_i} = \vee_{k=1}^{m} l_k \leq A_k \leq r_k$ in Eqn. 9. It can be transformed to the second case by performing a negation, i.e., $\bar{P}_{S_i}$=$P_{R_i} = \wedge_{k=1}^{m} l_k \leq A_k \leq r_k$, but with the selectivity $(1-\alpha)$.

Case 4. n>1 & m>1: The case is that a predicate has several *clause*s with more than one *literal*s in Eqn. 9. Since our target is to guarantee the selectivity α, we draw inspiration from *Set Theory* (i.e., **Rule₁** and **Rule₂**) to lower down the complexity by reducing instantiating parameters [27, 32] with domain boundary values.

$$\textbf{Rule}_1 : literal_i \cup literal_j = literal_j \quad if \quad literal_i \longleftarrow \emptyset$$
$$\textbf{Rule}_2 : clause_i \cap clause_j = clause_j \quad if \quad clause_i \longleftarrow U$$

Suppose the example predicate component of the $literal/clause$ is formatted as $l_k \leq A_k \leq r_k$. $Rule_1$ tries to reduce the literals by instantiating them to obtain an empty result $\emptyset$, which can be accomplished by assigning l_k the value of the domain + ϵ ($\epsilon > 0$) or r_k the value of the domain - ϵ (see Table. 3). Similarly, $Rule_2$ can reduce some clauses by assigning parameters with the domain boundaries to produce a universal set U. Based on $Rule_1$ and $Rule_2$, we can successfully reduce the complex predicates into the following two simple cases, which are $n > 1$ and $m = 1$ or $n = 1$ and $m > 1$. Take P_{S_i}=$\underline{(A_1{<}r_1 \vee A_4{>}l_2)} \wedge \underline{(A_2{+}A_3{<}r_3)}$ for example. It has two clauses (underlined). If we set r_1 in $clause_1$ to the maximum value of the domain A_1, P_{S_i} is simplified to $clause_2$.

To conclude, reservoir sampling can help to sample the dynamic data and populate the parameters in σ for a given selectivity efficiently. Meanwhile, based on the independence between join and selection predicates, we can freely pick a data range that satisfies the specified selectivity α under a stable distribution.

3.1.2 *Join Cardinality Control.*

Based on whether a selection predicate filters out new data, we categorize the join scenario into four cases as follows. We take the primary-foreign key (PK-FK) equi-join between the referenced relation R_i and referencing relation R_j to make a discussion with the data increasing as $\Delta\mathrm{R_i}$ and $\Delta\mathrm{R_j}$. This method can be generalized to other join types [54].

Case 1. $\Delta\mathrm{R_i}$=$\emptyset$ & $\Delta\mathrm{R_j}$=$\emptyset$**:** Join involves no new data. The join cardinality size is always the same, i.e., $|R_i \bowtie_{P_J} R_j|$=$|R_i' \bowtie_{P_J} R_j'|$.

Case 2. $\Delta\mathrm{R_i} \neq \emptyset$ & $\Delta\mathrm{R_j} = \emptyset$**:** Only the referenced relation R_i increases by $\Delta\mathrm{R_i}(\neq \emptyset)$. Due to the newly inserted primary keys in R_i' cannot join with any foreign keys in R_j, we can deduce its join result is the same as Case 1, i.e., $|(R_i \bowtie_{P_J} R_j) \cup (\Delta\mathrm{R_i} \bowtie_{P_J} R_j)|$=$|(R_i \bowtie_{P_J} R_j) \cup \emptyset|$=$|R_i \bowtie_{P_J} R_j|$.

Case 3. $\Delta\mathrm{R_i} = \emptyset$ & $\Delta\mathrm{R_j} \neq \emptyset$**:** Only the referencing relation R_j increases by $\Delta\mathrm{R_j}(\neq \emptyset)$. Since the newly inserted keys follow the same distribution as the initial keys, it ensures that the increment join cardinality from $\Delta\mathrm{R_j}$ is proportional to R_j, i.e., $|(R_i \bowtie_{P_J} \Delta\mathrm{R_j})|$=$\frac{|\Delta\mathrm{R_j}|}{|R_j|}|(R_i \bowtie_{P_J} R_j)|$. Then we have the following equation.

$$|(R_i \bowtie_{P_J} R_j) \cup (R_i \bowtie_{P_J} \Delta\mathrm{R_j})| {=} |(R_i \bowtie_{P_J} R_j)| {+} |(R_i \bowtie_{P_J} \Delta\mathrm{R_j})|$$
$$= |(R_i \bowtie_{P_J} R_j)| {+} \frac{|\Delta\mathrm{R_j}|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| {=} (1{+}\frac{|\Delta\mathrm{R_j}|}{|R_j|})|(R_i \bowtie_{P_J} R_j)|$$

In Case 3, the join cardinality change is linearly related to $\Delta\mathrm{R_j}$, which can be quantified by the throughput of T-engine.

Case 4. $\Delta\mathrm{R_i} \neq \emptyset$ & $\Delta\mathrm{R_j} \neq \emptyset$**:** Both the referenced and referencing relations increase. The join cardinality can be first deduced based on *Distributive Law* as in Eqn. 10.

$$\begin{aligned} &|(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} (R_j \cup \Delta\mathrm{R_j})| \\ &= |R_i \bowtie_{P_J} R_j| + |\emptyset| + |R_i \bowtie_{P_J} \Delta\mathrm{R_j}| + |\Delta\mathrm{R_i} \bowtie_{P_J} \Delta\mathrm{R_j}| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} \Delta\mathrm{R_j}| \end{aligned} \tag{10}$$

The cardinality increment is from $|(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} \Delta\mathrm{R_j}|$, where the number of foreign keys in $\Delta\mathrm{R_j}$ can come from either R_i or $\Delta\mathrm{R_i}$, represented by $|\Delta\mathrm{R_j}^o|$ and $|\Delta\mathrm{R_j}^n|$ respectively, and $|\Delta\mathrm{R_j}| = |\Delta\mathrm{R_j}^o| + |\Delta\mathrm{R_j}^n|$. So, Eqn. 10 can be further transformed into Eqn. 11.

$$\begin{aligned} &|R_i \bowtie_{P_J} R_j| + |(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} \Delta\mathrm{R_j}| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} (\Delta\mathrm{R_j}^o \cup \Delta\mathrm{R_j}^n)| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \bowtie_{P_J} \Delta\mathrm{R_j}^o) \cup \emptyset \cup (\Delta\mathrm{R_i} \bowtie_{P_J} \Delta\mathrm{R_j}^n) \cup \emptyset| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \bowtie_{P_J} \Delta\mathrm{R_j}^o) \cup (\Delta\mathrm{R_i} \bowtie_{P_J} \Delta\mathrm{R_j}^n)| \end{aligned} \tag{11}$$

For $\Delta\mathrm{R_j}^o$ is only related to R_i and it is semantics consistent with R_j in data distribution, $|(R_i \bowtie_{P_J} \Delta\mathrm{R_j}^o)|$ has the same result as Case 3, i.e., $|(R_i \bowtie_{P_J} \Delta\mathrm{R_j}^o)| = |\frac{|\Delta\mathrm{R_j}^o|}{|R_j|}(R_i \bowtie_{P_J} R_j)|$, which is linearly related to the original join cardinality. To obtain a stable increasing of $|\Delta\mathrm{R_i} \bowtie_{P_J} \Delta\mathrm{R_j}^n|$, an *equal-expansion-ratio-restricted joint distribution* between R_i' and R_j' can guarantee our expectation as defined in Eqn. 12, which is popularly adopted by current OLAP benchmarks [51].

$$\frac{|\Delta\mathrm{R_i}|}{|R_i|} = \frac{|\Delta\mathrm{R_j}^n|}{|R_j|} = \frac{|\Delta\mathrm{R_i} \bowtie_{P_J} \Delta\mathrm{R_j}^n|}{|R_i \bowtie_{P_J} R_j|} \tag{12}$$

Combine Eqns. 11 and 12 and then we have

$$\begin{aligned} &\frac{|\Delta\mathrm{R_j}^o|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| + \frac{|\Delta\mathrm{R_j}^n|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| \\ &= \frac{|\Delta\mathrm{R_j}^o| + |\Delta\mathrm{R_j}^n|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| = \frac{|\Delta\mathrm{R_j}|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| \end{aligned} \tag{13}$$

Finally, for Case 4, it has a linear increase as Eqn. 14.

$$|(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} (R_j \cup \Delta\mathrm{R_j})| = (1{+}\frac{|\Delta\mathrm{R_j}|}{|R_j|})|R_i \bowtie_{P_J} R_j| \tag{14}$$

To conclude, for the referenced/referencing relation R_i/R_j under the premise of the semantic consistent changes in data distribution, join cardinality caused by T-engine can be well quantified and follow a linear increasing.

3.1.3 *Projection and Aggregation Cardinality Control.*

Since the projection operator Π focuses on selecting specific attributes and has no impact on cardinalities; and the 'DISTINCT' operator's deduplication is usually covered in some aggregation operators, e.g., 'GROUP BY', we then exploring the cardinality control for the aggregation operators. We group the *aggregation* $\mathcal{G}$ into two classes according to the computational complexity. The first class is related to the input cardinality **s**ize (i.e., $card_s$), e.g., COUNT and SUM, while the second one is related to the **d**istinct cardinality of input (i.e., $card_d$), e.g., GROUP BY. Since $\mathcal{G}$ usually happens after *join* and *selection*, $card_s$ of $\mathcal{G}$ has been well controlled with a stable change guaranteed by the cardinality control methods to *selection* (in §3.1.1) and *join* (in §3.1.2). However, we still cannot guarantee the distinct values $Card_d$ in the input $Card_s$ for $\mathcal{G}$, which may greatly influence the aggregation computational complexity of the second case, which is related to distinct values. Notice that aggregation on non-key column $\mathcal{G}_{nk}$ usually involves category data of various types, e.g., date and country. For example, the $Card_d$ of column $OrderPriority$ in TPC-H is only 5. Then the categorized data has a limited impact on performance [27]. Aggregation on key columns generally happens on foreign keys (FKs) denoted as $\mathcal{G}_{fk}$, since primary keys (PKs) are distinct and used as the identifiers of rows. Note that the aggregation on the FK column ($\mathcal{G}_{fk}$) is independent of the selection on non-key columns, we can directly deduce the cardinality of the $\mathcal{G}_{fk}$ operator by Eqn. 15. Theoretically, $Card_d$ of the FK column is dependent on PK column of the referenced relation, which is unpredictable. Fortunately, the cardinality of $\mathcal{G}_{fk}$ is equal to the size of joinable PKs of the referenced table, which can be achieved by a left semi-join ($\ltimes_{P_J}$)[55]. Finally, it is represented by Eqn.16.

$$
\begin{aligned}
&\mathcal{G}_{fk}((R_i \cup \Delta\mathrm{R}_\mathrm{i}) \bowtie_{P_J} (R_j \cup \Delta\mathrm{R}_\mathrm{j})) && (15)\\
&= (R_i \cup \Delta\mathrm{R}_\mathrm{i}) \ltimes_{P_J} ((R_i \cup \Delta\mathrm{R}_\mathrm{i}) \bowtie_{P_J} (R_j \cup \Delta\mathrm{R}_\mathrm{j})) \\
&= (R_i \cup \Delta\mathrm{R}_\mathrm{i}) \ltimes_{P_J} (R_j \cup \Delta\mathrm{R}_\mathrm{j}) \\
&= |R_i \ltimes_{P_J} R_j| + |(R_i \ltimes_{P_J} \Delta\mathrm{R}_\mathrm{j}{}^{o}) \cup (\Delta\mathrm{R}_\mathrm{i} \ltimes_{P_J} \Delta\mathrm{R}_\mathrm{j}{}^{n})| \\
&= |R_i \ltimes_{P_J} R_j \cup (R_i \ltimes_{P_J} \Delta\mathrm{R}_\mathrm{j}{}^{o})| + |(\Delta\mathrm{R}_\mathrm{i} \ltimes_{P_J} \Delta\mathrm{R}_\mathrm{j}{}^{n})| \\
&= |R_i \ltimes_{P_J} R_j| + |\Delta\mathrm{R}_\mathrm{i} \ltimes_{P_J} \Delta\mathrm{R}_\mathrm{j}{}^{n}| && (16)\\
&= |R_i \ltimes_{P_J} R_j| + \frac{|\Delta\mathrm{R}_\mathrm{i}|}{|R_i|}|R_i \ltimes_{P_J} R_j| = (1+\frac{|\Delta\mathrm{R}_\mathrm{i}|}{|R_i|})|R_i \ltimes_{P_J} R_j| && (17)
\end{aligned}
$$

Since we have followed the *equal-expansion-ratio-restricted joint distribution* requirement to the increasing of join tables, we have $(\Delta\mathrm{R}_\mathrm{i} \ltimes_{P_J} \Delta\mathrm{R}_\mathrm{j}{}^{n}) = \frac{|\Delta\mathrm{R}_\mathrm{i}|}{|R_i|}|R_i \ltimes_{P_J} R_j|$. A linear expansion of $Card_d$ on FKs can still be achieved, as represented by Eqn. 17.

3.2 Control of Data Access Distribution

Another key point to keep a consistent computational complexity is maintaining the consistency of data access distribution [43].

For selection operators, suppose the generated parameters denoted as l_k^t and r_k^t, i.e., left/right boundaries at time t for a query template. We propose to take a *pairwise-step-expansion* to boundary parameters for its subsequent generation. That is its newly generated parameters l_k^{t+1} and r_k^{t+1} at time $t+1$ cover all the data accessed in the previous query by an equal expansion step size s to both boundary parameters l_k^t and r_k^t. Specifically, for a selectivity α of a selection, we have $l_k^{t+1} = l_k^t - s$ and $r_k^{t+1} = r_k^t + s$, with r_k^{t+1} $=l_k^{t+1} + N^{t+1} * \alpha$, N^{t+1} as the size of relation at time $t+1$ and s= $\alpha/2 \cdot (N^{t+1} - N^t)$. Follow this way, we regulate parameters from the same initial ones of a query template, the consistent data access distribution is guaranteed under the same OLTP throughput.

For join and projection/aggregation operators, the cardinality is linear related to original relations, which are $(1+\frac{|\Delta\mathrm{R}_\mathrm{j}|}{|R_j|})|R_i \bowtie_{P_J} R_j|$ and $(1+\frac{|\Delta\mathrm{R}_\mathrm{i}|}{|R_i|})|R_i \ltimes_{P_J} R_j|$. We can further simplify them as $|R_j + \Delta\mathrm{R}_\mathrm{j}|$ and $|R_i + \Delta\mathrm{R}_\mathrm{i}|$. Since $\Delta\mathrm{R}_\mathrm{i/j}$ follows the same distribution as its original relation $R_{i/j}$, we can ensure the consistent access distribution. More discussion can be found in technical report [54].

Finally, query complexity is controllable, and it facilitates the benchmarking repeatability for HTAP systems.

3.3 Data Size-based Latency Modeling.

We have achieved a stable linear increase of the cardinality for each SPJA operator in a query and the consistent access distribution as data grows. For fairly benchmarking HTAP systems, it's crucial to compare performance under the same data size. However, even with the same throughput from T-engines, two obstacles still exist in performance benchmarking. Firstly, query start time is uncertain due to the variant processing ability of A-engines, which causes accessing different data sizes. So comparison based on the **average latencies is unfair**. Secondly, A-engines of different HTAP systems following different data sharing models can see **different sizes of data** at the same time point. Under the premise of consistent data access distribution, the execution cost of a query is closely tied to the cardinality of each operator [28], and the algorithm complexity of most operators is $O(M)$ or $O(MlogM)$, with M as the input cardinality [33]. It then motivates us to build a log-linear regression model $L(S) = a \cdot Slog(S) + b \cdot S + c$ to predicate query latency, where $L(S)$ is the query latency under the data size S, and a, b and c are three fitting parameters.

To obtain the real-time data size S for A-engine without costly scan in relations, we propose to create transaction synchronization tables ($SYNTab$) on both sides of the T- and A-engines denoted as $SYNTab^T$ and $SYNTab^A$ as shown in Fig. 6. In $SYNTab^T$, it records the number of transactions (Trx) completed, and in $SYNTab^A$, it presents the number of modifications (from Trx) that have been synchronized. Transactions synchronized to $SYNTab^A$ generally lag behind the ones executed in $SYNTab^T$ under weak data consistency. For example, in Fig. 6, there are three types of transactions, i.e., Trx_1, Trx_2, Trx_3, among which only 8 among 10 Trx_1 have been synchronized to A-engine. Based on the $SYNTab$, it is easy to predicate the incremental data after the synchronization based on the transaction semantics.

In practice, $SYNTab$ is a lightweight relation in T-engine which is updated when a new transaction is committed. To reduce contentions, one relation can be implemented for one type of Trx which stores one single number, which has low storage cost and can be located in memory. Since writes in transactions (e.g., TPC-C) usually have complex semantics, updating $SYNTab$ by a single point write is much faster than the other operations and will not block the normal execution of transactions. Meanwhile, as the updates in $SYNTab^T$ will be synchronized to A-engine under the same data sharing model of the HTAP system, and every relation is read under the same snapshot timepoint [31], we can employ the information in $SYNTab^A$ to calculate the data size increasing for the related queries in A-engine. For modeling, to avoid possible performance fluctuations from network or other factors, we launch the regression fitting approach based on *Huber Loss* function (defined in Eqn. 18), which is insensitive to noise and serves as an effective method for deciding fitting parameter of a, b and c in $L(S)$ [6, 16, 24].

$$
Huber\ Loss(\theta, \mu) = \begin{cases} \frac{1}{2}\mu^2, & \text{if } |\mu| \leq \theta, \\ \theta(|\mu| - \frac{1}{2}\theta), & \text{otherwise.} \end{cases} \tag{18}
$$

Specifically, $\mu = L(S) - \hat{L}(S)$ represents the residual between real latency $L(S)$ and predict latency $\hat{L}(S)$ on data size S, and θ is an adjustable hyperparameter. Therefore, For each query q_i, its query latency with data size S_{ik} is L_{ik}. Each pair of L_{ik} and S_{ik}

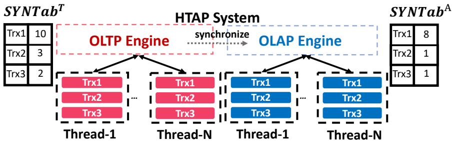


Figure 6: Data Size and Query Latency Modeling

can be used to fit the best a_i, b_i, and c_i for q_i which expects to provide the least residual between $L_i(S)$ and $\hat{L}_i(S)$. In a workload with W queries, we finally have W individual fitting functions. Finally, for the size of concurrency clients TH in A-engine, we take $QphH$ [50, 51] to demonstrate the overall OLAP performance, which is $QphH = 3600 * \frac{W*TH}{\sum_i^W L_i(S)}$. In $QphH$, the W queries are taken as a whole and we accumulate their overall latency $L_i(S)$ based on the specific data size S. So every system under test can be compared by the latency under the specific same data size S.

4 QUERY-ORIENTED DATA FRESHNESS

4.1 Data Sharing Models

As we have mentioned in § 2.1.2, there is a distinct trade-off between performance and freshness in HTAP systems. Before we give a formal definition of freshness, we formalize the popularly used data sharing models (*DSMs*) in current HTAP systems, which helps to understand the subsequent discussion for freshness measurement.

4.1.1 Version Evolution of Data. An HTAP system consists of both T- and A-engine. To introduce the data sharing between the two engines, we give an example in Fig. 7. For simplicity, each transaction *Trx* contains only a single read or write operation, and there are 8 clients currently accessing the systems, each of which corresponds to a session. Specifically, $Client_1$ and $Client_2$ launch 4 write *Trx*s, i.e., $w_1 \cdots w_4$, and the other clients launch read *Trx*s. Before we define *DSM*, we introduce some concepts for version evolutions.

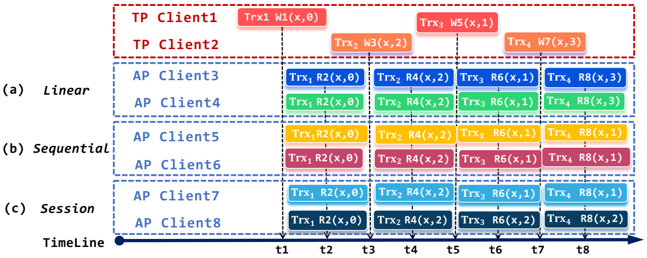


Figure 7: Read/Write x under Different Data Sharing Models

Version Write: The write operation $w_i(x, v)$ (simplified as w_i=v) denotes the action of a committed transaction *Trx* writing a data item x to value v. The commit time is marked as t_i.

Example 1. In Fig. 7, $Client_1$ executes two write Trx_1, Trx_3 to x and updates x to 0 and 1 at time t_1 and t_5 respectively.

Version Write Sequence: Version write sequence S_t refers to the full linear write sequence to x, which covers all sequential version writes along the timeline with version write time no larger than time t, i.e., $S_t = \{w_1, \cdots, w_i, \cdots, w_j, \cdots, w_k\}$ s.t., $\forall i < j, t_i < t_j$, $t_k = t$, and $\forall w_n, w_{n+1}, \nexists w_l : t_n < t_l < t_{n+1}$.

Example 2. In Fig. 7, the version write sequence S_8 at t_8 is $\{w_1, w_3, w_5, w_7\}$. Concretely, the evolution of item x is 0->2->1->3.

Prefix Write Sequence: The prefix write sequence S_t^p is a linear full sub-sequence of the version write sequence to x before time t, i.e., $S_t^p = \{w_1, w_2, w_3, ..., w_p\}$, s.t., $S_t^p \subseteq S_t, \forall i < j, t_i < t_j$ and $t_p \leq t$, and $\forall w_i, w_{i+1} \in S_t^p, \nexists w_j (\in S_t) : t_i < t_j < t_{i+1}$.

Example 3. For $S_8 = \{w_1, w_2, w_3, w_4\}$ at t_8, its prefix write sequence can be $S_8^{p_1} = \{w_1\}$, $S_8^{p_2} = \{w_1, w_2\}$, $S_8^{p_3} = \{w_1, w_2, w_3\}$, $S_8^{p_4} = \{w_1, w_2, w_3, w_4\}$.

Data Version: A data version v_t of x is the result of executing a prefix write sequence S_t^p. The version-stamp of x at t is defined by the length of its prefix write sequence, i.e., $|S_t^p|$.

Example 4. For $Client_3$ at t_6, the result of executing $S_6^p = \{w_1, w_2, w_3\}$ on x is 1, which is a prefix sequence of S_8. The data version and data version-stamp is v_6=1 and $|S_6|$=3, respectively.

Version Read: For a read $r_t(x, v)$ of a session (simplified as r_t=v) at time t, it has the monotonicity of its read sequence, i.e., $\forall t_i \leq t_j, |S_i| \leq |S_j|$.

Example 5. For $Client_3$, Txn_3 reads the data version of x at t_6, i.e., r_6=1, from a write sequence S_5, while r_4=2 is from its prefix write sequence S_3, among which $S_3 \subseteq S_5$.

4.1.2 Data Sharing Model. Suppose for data x, its latest write is w at time $w.t$ and its write version and write sequence are $v_{w.t}$ and $S_{w.t}$; a read r at time $r.t$ from session s reads the version of x, i.e., read version $v_{r.t}$, from a prefix write sequence $S_{r.t}$, with $S_{r.t} \subseteq S_{w.t}$ and $r.t \leq w.t$. The data sharing model (*DSM*) defines the data version gap of between T- or A-engines. Specifically, DSM_{TA} (resp. DSM_{AA}) in Eqn. 19 declares the version gap between T- (from write sequence) and A-engines (from read sequence) (resp. between read sequences of $r_{i/j}$ from different sessions of A-engines). Based on the differences of the data version gaps, represented by p and q in Eqn. 19, we summarize three *DSMs* as followings.

$$\begin{aligned} DSM_{TA}(v_{w.t}, v_{r.t}) &= |S_{w.t}|\text{-}|S_{r.t}|\text{=}p, \quad s.t., \ w.t\text{=}r.t \\ DSM_{AA}(v_{r_i.t}, v_{r_j.t}) &= |(|S_{r_i.t}| - |S_{r_j.t}|)|\text{=}q, \quad s.t., r_i.t\text{=}r_j.t \end{aligned} \tag{19}$$

Linear Consistency refers to the data accessed on A-engines has exactly the same version as the data on T-engines. It ensures that for any read r at time $r.t$, its read version $v_{r.t}$ must be produced by all completed OLTP writes in $S_{r.t}$ before r initiated, i.e., p = 0. Therefore, all A-engines read the same data at time $r.t$, i.e., q = 0.

Sequential Consistency ensures that for any read operations even from different sessions at time $r.t$, all read versions are produced by the same prefix write sequence of $S_{r.t}^p$, i.e., $p \geq 0, q = 0$.

Session Consistency ensures that for any read operation r from a session s at time $r.t$, its read data version $v_{r.t}^s$ is produced by executing a prefix write sequence of $S_{r.t}^s$ which has its version-stamp no smaller than the ones of any old read at $r'.t$ ($r'.t \leq r.t$) from the same session s, i.e., $p \geq 0, q \geq 0$.

These three *DSMs* are supported by various HTAP systems designed for various application scenarios as summarized in Table 4.

Intuitively, data freshness refers to how up-to-date the accessed data by A-engine is [4], and in previous work, data freshness is typically measured by three approaches. First, it can be calculated

Table 4: Version Gaps of Different Data Sharing Models

Data Sharing Model				
Consistency Level		Linear	Sequential	Session
Gaps	p	0	≥ 0	≥ 0
	q	0	0	≥ 0
Representative HTAP Systems		TiDB [17] OceanBase [61] Oracle [22]	F1 Lightning [59] BatchDB [30] Vegito [47]	PolarDB [5] Aurora [52] Hyper [19]

by the overlap ratio of identical tuples between the T- and A-engines [45]. However, it's expensive to scan and compare the whole sdatabase. Second, in Vegito [47] and ByteHTAP [7], freshness is calculated based on the time elapsed from the latest data version generated by T-engine to its synchronization to A-engine [4, 9]. Finally, HATtrick [31] proposes to measure freshness by the global snapshot lag time between the **f**irst **n**on-**s**een transaction commit time t^{fns} by a query q of A-engine at its query start time t_q^s, i.e. $freshness = max(0, t_q^s - t^{fns})$. Specifically, to check the first non-seen transaction, it maintains a freshness table to record the latest transaction ID (monotonically increase) for each T-client. Each transaction write will include an additional update to the freshness table with the transaction ID as well as its commit time on the client side. Upon starting an OLAP query, it scans all freshness tables in both T- and A-engines to find the first non-seen transaction commit time by the transaction IDs it can observe. Both of the last two methods focus on the global data consistency instead of the user interested data.

4.2 Query-oriented Freshness

In real-world business scenarios, users are concerned with the queried data, which should satisfy the expected freshness constraints [4, 7], instead of the overall global consistency of the entire system. It means that users do not merely care about the global data consistency between T- and A-engines, but the data queried. This inspires us to benchmark freshness in the granularity of the query and then $Vodka$ proposes to measure freshness by collecting the local snapshot consistent time duration based on the data accessed in a query q, defined in Eqn. 20, where t_q^s and t_q^{lsc} are the query start time and the **l**ocal **s**napshot **c**onsistent time for q.

$$freshness(q) = max(0, t_q^{lsc} - t_q^s) \tag{20}$$

Specifically, supposing q involves a set of data D_q and TXN_q is the collection of transactions writing to data in D_q, the last impactful transaction (L-Txn_q) is defined as the transaction that writes D_q with the largest timestamp and has the highest probability of causing inconsistency between T- and A-engines. The perfect snapshot consistency for q means that the modification from L-Txn_q is applied/synchronized to A-engine in realtime, i.e., $freshness(q)$=0. To expose and measure the data consistency of D_q, we add an anchor column called *version* for every relation to denote the number of times been modified to each row. For example, when a row is inserted, *version* is initialized to 1 which monotonically increases for any modification. Thus, when a query q starts at time t_q^s, we locate the last impactful transaction associated with the row *version*s, which are aggregated in both T- and A-engines, represented by $ver_{tp}(t_q^s)$ and $ver_{ap}(t_q^s)$ respectively. Since T-engine generates new versions to A-engine, we always have $ver_{tp}(t_q^s) \geq ver_{ap}(t_q^s)$. For D_q, if the version aggregation of $ver_{ap}(t_q')$ is no smaller than $ver_{tp}(t_q^s)$ ($t_q' \geq t_q^s$), A-engine must cover all the latest versions at the query start time t_q^s on T-engine in view of Theorem. 6. We define t_q^{lsc} as the first time to have $ver_{tp}(t_q^s) \leq ver_{ap}(t_q^{lsc})$. So if the version aggregation of A-engine is still no larger than that of T-engine, i.e., $ver_{ap}(t_q') < ver_{tp}(t_q^s)$, $A-$engine will kept launching the aggregation query until t_q^{lsc} and t_q^{lsc} is taken as the consistency duration for query q.

Theorem 6. *Supposing any two rows, e.g., r_i, r_j, are accessed by q, the versions seen in T- and A-engines at t_q^s are $ver_{tp}^{r_i}(t_q^s)$, $ver_{tp}^{r_j}(t_q^s)$, $ver_{ap}^{r_i}(t_q^s)$, and $ver_{ap}^{r_j}(t_q^s)$. If at time $t(\geq t_q^s)$, $ver_{ap}^{r_i}(t) + ver_{ap}^{r_j}(t) \geq ver_{tp}^{r_i}(t_q^s)+ver_{tp}^{r_j}(t_q^s)$, then we have $ver_{ap}^{r_i}(t) \geq ver_{tp}^{r_i}(t_q^s)$ and $ver_{ap}^{r_j}(t) \geq ver_{tp}^{r_2}(t_q^s)$.*

Proof. Even by different data sharing models in HTAP systems, for a query at time t_q^s, the latest data version r in T-engine is no older than the one visited by the A-engine, i.e., $ver_{tp}^r(t_q^s) \geq ver_{ap}^r(t_q^s)$.

Modifications in T-engine will always increase the data version, so if $t_k > t_m$, we always have
$ver_{tp}^r(t_k) \geq ver_{tp}^r(t_m)$ and $ver_{ap}^r(t_k) \geq ver_{ap}^r(t_m)$.

Therefore, for the version aggregation of r_i and r_j in T-engine and A-engine at t_q^s, we have $val_{tp}(t_q^s) \geq val_{ap}(t_q^s)$ with
$val_{tp}(t_q^s)$ =$ver_{tp}^{r_i}(t_q^s)$+$ver_{tp}^{r_j}(t_q^s)$ and $val_{ap}(t_q^s) = ver_{ap}^{r_i}(t_q^s)$+$ver_{ap}^{r_j}(t_q^s)$.

Supposing data sharing model in HTAP systems can provide a globally consistent snapshot for all local data in the T- and A-engine, we have that if $ver_{ap}^{r_i}(t_k) \geq ver_{tp}^{r_i}(t_m)$, then $ver_{ap}^{r_j}(t_k) \geq ver_{tp}^{r_j}(t_m)$. Thus, for any $t \geq t_q^s$, if $ver_{ap}^{r_i}(t) + ver_{ap}^{r_j}(t) \geq ver_{tp}^{r_i}(t) + ver_{tp}^{r_j}(t)$, it implies that $ver_{ap}^{r_i}(t) \geq ver_{tp}^{r_i}(t_q^s)$ or $ver_{ap}^{r_j}(t) \geq ver_{tp}^{r_j}(t_q^s)$, correspondingly we have $ver_{ap}^{r_j}(t) \geq ver_{tp}^{r_j}(t_q^s)$ or $ver_{ap}^{r_i}(t) \geq ver_{tp}^{r_i}(t_q^s)$. That is the versions in A-engine at time t must cover the versions in T-engine at time t_q^s.

The theorem is proved. □

4.3 Freshness Quality

Freshness Fail Rate. It has been declared that different businesses have their own data freshness requirements, i.e., *freshness threshold*. For instance, an Internet of Things scenario prioritizes data freshness within 200ms, while online gaming requires 100ms [47]. Therefore, to benchmark the adaptability of HTAP systems for different application scenarios, we define *freshness fail rate* as a metric to expose the ratio of queries that have not guaranteed the expected freshness requirement. Let $\mathcal{T}$ represent the business freshness demand. If $freshness(q) > \mathcal{T}$, we mark q fail and put it into Q_{fail}. We finally collect and calculate *freshness fail rate* by F=$\frac{|Q_{fail}|}{|Q|}$.

Unified Measurement for HTAP Systems. Due to the existence of different data sharing models, quick response of queries, i.e., low latency to clients, may benefit from the low degree of data consistency between T- and A-engines, i.e., low freshness or fewer data. It then biases customers from systems providing strict consistency models, e.g., the linear one, for its high cost of synchronization. To compare HTAP systems fairly, we propose a *unified linearly-consistent* measurement $Latency^u(q)$ and it aligns all systems to the strict linear data sharing model by $Latency^u(q) = freshness(q) +$

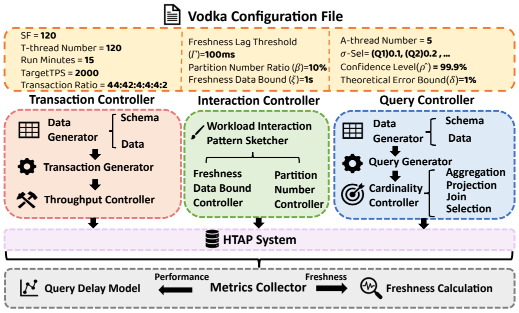


Figure 8: Vodka Architecture Overview

$latency(q)$ among which $freshness(q)$ and $latency(q)$ are the synchronization latency and the query response latency respectively.

In this way, performance and freshness are orchestrated to provide an overall evaluation metric for real-time queries among different HTAP systems.

4.4 Quantify Data Sharing Pressure

In HTAP systems, different amount of fresh data accessed by A-engines will cause distinct degrees of data sharing pressure. Since more accessed fresh data introduces larger synchronization costs and resource consumption in network bandwidth, or CPU, there is usually a trade-off between the performance and freshness in HTAP systems [45]. Due to the new data from T-engine is usually synchronized to A-engine in a parallel partition-oriented way [17, 52]. Suppose there are P partitions based on the partition rule, e.g., hash or range, involved by modifications from T-engine during the time boundary of freshness requirement ξ. *ξ-bounded workload interaction pattern* quantifies the amount of fresh data access between T- and A-engines within a freshness data bound ξ.

Definition 1. *ξ-bounded workload interaction pattern: It declares the ratio of partitions β ($0 \leq \beta \leq 1$) queried by A-engines w.r.t the P partitions modified by $T-$engine within time range ξ, i.e., expecting to access $\beta \times P$ fresh data partitions.*

So we can quantify the data sharing pressure by specifying the workload interaction pattern with different ξ and β even having different partition rules (Implementation details in § 5).

5 VODKA IMPLEMENTATION

To adapt $Vodka$ for benchmarking HTAP systems, we construct an HTAP scenario based on TPC-C and TPC-H benchmarks following previous work [10, 11], but address the existing limitations during combination. In this section, we introduce the benchmarking architecture of $Vodka$ (§ 5.1), and give a brief overview of the distinct implementations for a consistent HTAP scenario (§ 5.2).

5.1 Implementation Architecture of Vodka

In Fig. 8, we show the implementation architecture of $Vodka$. In *configuration file*, we specify the number of warehouses (*SF*) for *Data Generator*, the number of T-threads for *Transaction Generator* and the target TPS controlled by *Throughput Controller*. For A-engine, we can set the number of A-threads for *Query Generator*. Note that based on selection-selectivity and the incremental size of relations, we can then deduce the cardinality from $\bowtie$ and σ operators (as in § 3) in *Cardinality Controller*. Specifically, *σ-Sel* is calculated as the proportion of rows filtered by selection predicate P_S to the original relation R under default parameters. For example, for Q1 in TPC-H, its *σ-Sel* is calculated by *'select count(*) from OrderLine where ol_delivery_d<= 1993-09-02 00:00:00 / select count(*) from OrderLine'*, which controls to instantiate parameters in Q1 under continuous modifications from T-engine. *Confidence Level*($\hat{\rho}$) and *Theoretical Error Bound* (δ) are used to determine the reservoir sampling size. *Freshness Lag Threshold* ($\mathcal{T}$) is to quantify the expected data freshness and to calculate *freshness fail rate* (as in § 4.3); *Partition Number Ratio* (β) and *Freshness Data Bound* (ξ) are specified to control the interaction pattern for hybrid workloads in *Workload Interaction Pattern Sketcher*, i.e., determine the range of fresh data to read (as in § 4.4). Finally, we collect the result by *Metrics Collector* to expose performance and freshness.

5.2 HTAP Scenario Synthesis

As we have analyzed at the beginning of this section that to build a semantically consistent scenario for HTAP systems, the adjustment to either schema/data or workload in TPC-C/-H is imperative.

5.2.1 Data/Schema Consistency. For the schema, we follow CH-benchmark [11] by replacing the relations $PartSupp$, $Part$ and $LineItem$ in TPC-H with $Stock$, $Item$ and $OrderLine$ in TPC-C, which have almost the same application semantics. Tables with the same semantics have the schema by unifying the attributes from both TPC-C and TPC-H (details in Technical Report [54]). Since the performance of A-engine is high related to data distribution, we then rule data distribution of TPC-C in accordance with TPC-H. New relation $SYNTabs$ are created in both T-engine and A-engine to record the number of transaction writes by New-$Order$, $Payment$ and $Delivery$, and the synchronized status of the modifications.

5.2.2 Workload Consistency. Except the static dimension tables, e.g., $Country$, $Region$ and $Customer$, to guarantee semantic consistency of workload in HTAP systems, fact tables accessed by A-engines are supposed to read the modifications from T-engine. For instance, Q12 in TPC-H accesses the the order receiving time, i.e., *ol_receipt_d*. However, in the original TPC-C, this field is not appropriately modified, causing inconsistencies between the T- and A-engine for order processing, and newly delivered items may not be returned. To address this, we introduce a new transaction, *ReceiveGoods*, to complete the order lifecycle. *ReceiveGoods* simulates customer actions, including confirming received orders (*ol_receipt_d*), rejecting goods (*ol_return_flag*), and providing feedback (*o_comment*). To complete order processings, after a *Delivery* transaction ships an order, we update its *ol_delivery_d*. Subsequently, we update the $OrderLine$ relation with *ol_receipt_d* and *ol_return_flag*, and the $Order$ relation with *o_comment*.

5.2.3 Scenario Consistency. Since the real-time analysis of newly generated data by T-engine is the most attractive feature of HTAP systems, constructing queries to evaluate freshness is imperative in an HTAP scenario. $Vodka$ proposes to calculate query-oriented freshness by aggregating the column of $version$ to check real-time data consistency with SQL *FreshnessCheck* as following.

```
SELECT SUM(version),ol_delivery_d as da_anc FROM OrderLine
       WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
SELECT * FROM OrderLine WHERE ol_w_id BETWEEN ? AND ?
       AND ol_delivery_d In (da_anc,?,?,...))
```

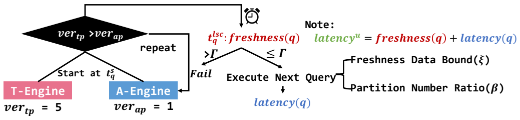


Figure 9: Freshness Check Process

Since the *OrderLine* relation meets the most frequent writes, *FreshnessCheck* is implemented to access *OrderLine*. Specifically, after a *Delivery* transaction commits, we can get the information for the latest updated order (called ***anchor** order*), including its *ol_w_id* (w_{anc})*, ol_d_id* (d_{anc})*, ol_o_id* (o_{anc}) and *ol_delivery_d* (da_{anc}). Then we fill these parameters into the first SELECT for SUM query of *FreshnessCheck* to check data consistency between T- and A-engines $w.r.t$ this *anchor order.* After submitting this version SUM query, it repeats the aggregation until A-engine observes all the newest versions from T-engine at time t_q^{lsc} as shown in Fig. 9. If consistency cannot be achieved during a target freshness threshold, i.e., *freshness(q)*> $\mathcal{T}$, we mark this query as a freshness fail; or else, we launch the second query in *FreshnessCheck.* It is used to calculate the query processing latency ($latency(q)$) by retrieving all the previously delivered orders within the specified freshness bound ξ $w.r.t$ *ol_delivery_d* of the *anchor order*, i.e., da_{anc} in *FreshnessCheck*, and partition key *ol_w_id* is taken to control the size of warehouses accessed, i.e., β. ξ and β sketch the workload interaction pattern. In practice, according to ξ, we maintain a fixed range of a circular array, which records the latest delivery date during the time interval of ξ and is updated once a *Delivery* transaction is committed. When we start *FreshnessCheck*, the array stops updating. For the second query in *FreshnessCheck*, we locate the *ol_delivery_d* of each order that falls in the circular array delivered before the delivery date da_{anc} of the *anchor order.* Note that following the process logic of TPC-C under *targetTPS* of T-engine, it is feasible to calculate the size of the array to store the delivered order date bound by time range ξ, which are all taken as candidates inside the 'IN' predicate of *FreshnessCheck.* Meanwhile, to control the interaction pattern between T- and A-engines, *Delivery* transactions are specified to write warehouses sequentially. Suppose each *Delivery* transaction writes K warehouses. By *targetTPS* and the ratio of delivery transactions (*Delivery*%), we can calculate the total number of warehouses modified by *Delivery* within time range of ξ as w_m =*targetTPS* $\cdot$ ξ $\cdot$ *Delivery*% $\cdot K$. For a target β, the size of the accessed warehouses is $w'_m = \lceil w_m \cdot \beta \rceil$, and we select and fill their *warehouseID* into the second query following Eqn. 21 with W as the total number of warehouses.

$$ol\_w\_id \in \begin{cases} [w_{anc} - w'_m, w_{anc}] & \text{if } w_{anc} \geq w'_m \\ [0, w_{anc}] \cup [W - w'_m + w_{anc}, W] & \text{if } w_{anc} < w'_m \leq W \end{cases} \quad (21)$$

6 EXPERIMENTAL EVALUATION

In this section, we run extensive experiments to illustrate the effectiveness of *Vodka* and explore the performance and freshness of three kinds of open-source HTAP systems, which are PostgreSQL Streaming Replication (PostgreSQL-SR) [41], OceanBase [60] and TiDB [17]. Specifically, PostgreSQL-SR relies on storage isolation, OceanBase relies on logical isolation, and TiDB's isolation method depends on its deployment mode. In our test, we deploy it with logical isolation. Before delving into the concrete experiments, we first describe the experimental configurations, the setup for each database, and the default argument settings.

Cluster configuration. Each system is deployed on the same three server cluster nodes, each of which has CentOS Linux release 7.9.2009, Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz with 96 physical cores, 375GB RAM and a 1.5TB pmem disk. Besides, *Vodka* is deployed independently in a client node with configurations of CentOS Linux release 7.9.2009, Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz with 48 physical cores, 265GB RAM. All nodes are connected via 10GbE network cards.

Database System configuration. For OceanBase(v4.2.0), each node is deployed to serve OLTP and OLAP tasks simultaneously. For TiDB(v7.1.0), its row store called TiKV and column store called TiFlash are deployed on each server by manually binding to half of the CPU cores [8, 30]. PostgreSQL-SR(v14.5) is deployed based on streaming replication [41]. We assign a single master node for OLTP tasks, a slave for disaster recovery backup and an asynchronous slave for OLAP tasks.

Benchmark configuration. By default, all experiments are performed on a data size of approximately 8GB, i.e., 120 warehouses with each thread binded to one warehouse [45, 63]. We set each system to have 16 intra-query parallel degree. The default reservoir sampling size is 40K rows (approximately 1MB) according to Hoeffding's Inequality with its theoretical error bound δ = 1% in a confidence level of $\hat{\rho}$ = 99.9%. By default, we set β=100% and ξ=5 second (s) to read all new data within 5s. Selectivity α is calculated from the initial round of queries from A-engine. During benchmarking, to avoid code-start problem, all experiments have 5 minutes (min) warm-up phase by default. To have a fair comparison, we run all experiments under the same stable OLTP throughput.

6.1 Evaluation of the Key Design of Vodka

We first conduct preliminary experiments for determining some experimental configuration parameters (§ 6.1.1). Then we launch experiments to verify *Vodka*'s key design (§ 6.1.2) and the fidelity of parameter fitting between data size and latency (§ 6.1.3).

6.1.1 Preliminary Experiments for Parameter Configuration. We first conduct an experiment to discover the maximum OLTP throughput (TPS) for three systems under 120 T-threads by employing one A-thread, which has a minimized impact on the performance of T-engine. The results reveal that PostgreSQL-SR achieves the highest TPS of 6.5 × 10^3 (6.5K) due to its standalone nature, that is centralized writing on a single node; OceanBase has a peak TPS of 5K, benefiting from its optimizations in LSM-Tree-based storage design and Paxos-based 2PC implementation [61]; TiDB has a relatively low TPS of 2K for introducing too many distributed transactions. To make a fair comparison among HTAP systems, we set the TPS of T-engines no larger than 2K. Additionally, to guarantee TPS=2K, we find the maximum number of A-threads is 10 for all the three systems. So, by default we set TPS=2K and A-threads=10.

6.1.2 Cardinality Control of OLAP Queries. We adopt *relative error*= $\frac{1}{|Q_{op}|}\sum_i \frac{||C_i|-|\hat{C}_i||}{|C_i|}$ [27] to measure the accuracy of cardinality control to all operators ($|Q_{op}|$) in queries Q, where $|C_i|$ and $|\hat{C}_i|$ are the runtime real and the predicted cardinalities of the i^{th} operator under a specified selectivity α. A smaller *relative error* means a better control. Since the cardinality estimation is only related to the throughput of T-engine and the parameter instantiation of TPC-H

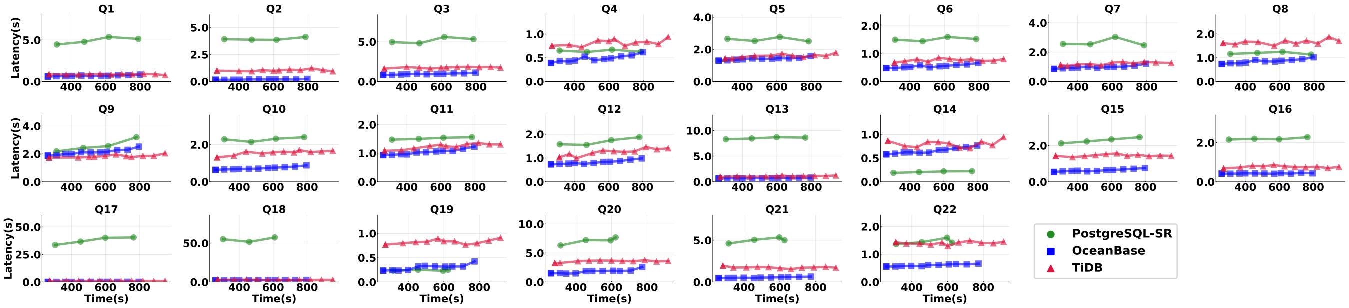


Figure 10: Query Latency *vs.* Data Size

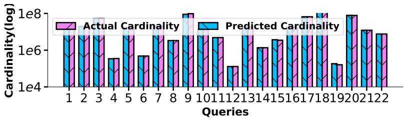
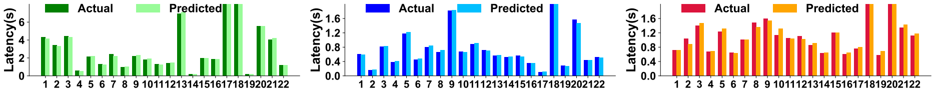


Figure 11: Relative Errors of Queries



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

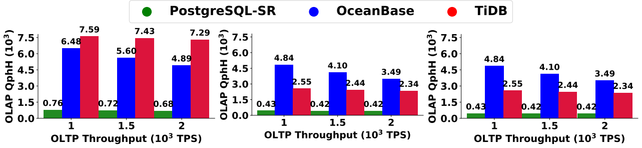
Figure 12: Actual & Predicted Latencies For All OLAP Queries

query templates, we take PostgreSQL-SR to illustrate *relative error* of our cardinality estimation by running OLTP workload with 2K TPS for 15 minutes, after which we run the whole OLAP workload and collect the cardinalities. The result in Fig. 11 demonstrates all queries have low relative errors (the maximum is 1.7% of Q19).

Meanwhile, we run TPC-C workloads and demonstrate OLAP query latency in Fig. 10 for Q1-Q22 with TPS=2K and 10 *A*-threads. As increasing test time (database size), our parameter instantiation method guarantees all latencies to expose approximately stable linear changes, which makes it feasible for modeling performance with increasing data size by our log-linear regression model (as in § 6.1.3). Note that these HTAP systems have performed differently for distinct queries. PostgreSQL-SR performs better than OceanBase or TiDB for queries Q8, Q14, Q19, and Q22, which benefit from the superior implementation of the Bitmap Index Scan (reduce scan cost). Besides, OceanBase demonstrates an overall better performance, especially for Q5, Q10, Q15, Q20, which benefit from its optimization in join order selection and performing nested-loop-join for small tables to reduce the size of intermediate results.

6.1.3 *Effectiveness of Log Linear Modeling to Latency.*

We validate the effectiveness of the latency modeling function $L(S) = a \cdot Slog(S) + b \cdot S + c$ under the default configuration. We learn parameters a, b and c for each query based on 10min performance results. We gather 100 pairs of the latency and the calculated data size every 6s based on the throughput of T-engine as the training set. We then decide θ of the *Huber Loss* function from the candidate set {0.01,0.1,1,10,100}, which have been widely adopted by previous work [23], to fit parameters by running an extra 5min to collect the new pairs as the validation set. We use Mean Squared Error (MSE) to compare modeling effects and select the best θ for each query. Then we continue running another 5min and sample the latency with its the calculated data size every 6s and get 50 pairs as test set. We compare the average actual latency with the average predicted latency in Fig. 12, and it shows our model peforms well with the maximum performance deviation < 5% in each system.



(a) 1 *A*-Thread (b) 5 *A*-threads (c) 10 *A*-threads

Figure 13: Performance of A-engine under Different TPS

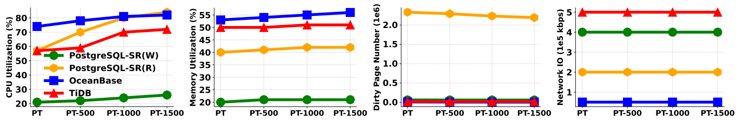
6.2 Comparison of HTAP Systems for Resource Isolation and Data Sharing

We benchmark the distinct features in different HTAP systems, including resource isolation in § 6.2.1 and data sharing in § 6.2.2.

6.2.1 *Resource Isolation.*

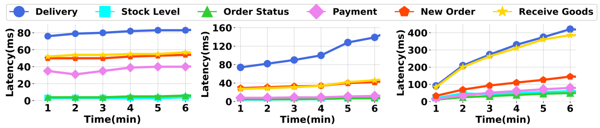
We demonstrate the OLAP performance (QphH) under different TPS by changing *A*-threads in Fig. 13.

For example in Fig. 13a, QphH of TiDB conquers the other two systems. When TPS=2K with 1 *A*-thread, QphH of TiDB is 1.5× and 6.8× higher than OceanBase and PostgreSQL-SR. It benefits from its optimization in data organization on TiFlash (*A*-engine) [38], e.g., column store, compression, vectorization execution, and so on. OceanBase has implemented vectorization query execution based on its row-column hybrid storage [2]. PostgreSQL-SR has no vectorization or column storage design, and the overall QphH of *A*-engine is inferior to the other two systems. We have observed an interesting phenomenon that as increasing TPS of T-engine, OceanBase experiences a relatively evident drop of per-thread QphH. This might be caused by OceanBase's MVCC mechanism, which meets longer version chains for larger writings. It then lowers OLAP scan efficiency. In contrast, TiDB, with its separate columnar replica and partition pruning in TiFlash, shows its performance is less sensitive to the change of data sizes. PostgreSQL-SR, due to its excellent bitmap index design, exhibits the least performance degradation as data size increases. Additionally, when increasing *A*-threads, all systems have per-thread QphH regressions. TiDB exposes the worst regression. This is primarily due to its severer multi-thread resource contention, especially CPU. Moreover, there is an optimization of *small table broadcasting* in OceanBase. It replicates tables with little updates across all servers, e.g., *Nation* in TPC-H. Then queries accessing these tables are localized and have a relatively low latency



(a) CPU (b) Memory (c) Dirty Page (d) Network IO

Figure 14: The Variation of Resource Utilization



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

Figure 15: Transaction Latency at Peak TPS *v.s.* *A*-threads

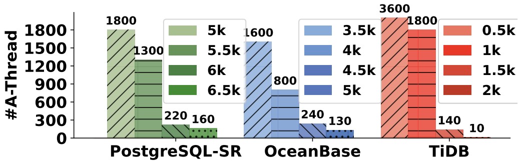


Figure 16: The Maximum A-thread Number

(relatively high QphH) when increasing *A*-threads. So *small table broadcasting* is worth considering for distributed HTAP Systems.

Moreover, since each system has a different throughput frontier by the given *T*-threads, resource contentions from *A*-engine vary under different OLAP pressures for they have different data sharing mechanisms. To demonstrate the resource isolation ability, we conduct tests by increasing OLAP threads for each system under various stable TPS of individual *T*-engine. High throughput of *T*-engine requires more resources in essence and is more sensitive to the intensive data sharing requirement from *A*-engine. The maximum TPS frontiers of the HTAP systems by our default *T*-engine setting have been given in § 6.1.1.

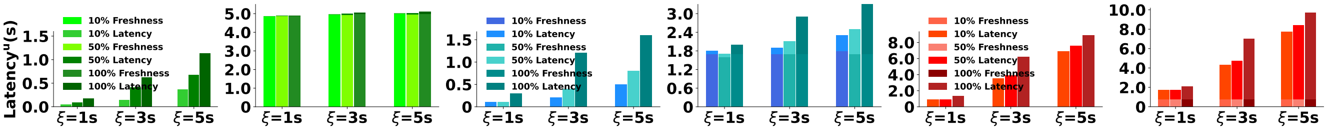
The maximum *A*-thread number of each stable TPS is shown in Fig. 16. PostgreSQL-SR provides the best resource isolation and it supports the maximum *A*-threads. For instance, for the highest TPS of PostgreSQL-SR, i.e., 6.5K, it can still run the most 160 *A*-threads and when TPS drops to 5K, it can accommodate nearly 1800 *A*-threads. In contrast, OceanBase can support fewer *A*-threads (130) under its highest TPS, i.e., TPS=5K. The difference arises from the fact that PostgreSQL-SR has the best storage isolation, while OceanBase can only support a logical isolation by assigning CPU resources to different workloads. TiDB provides relatively worse isolation and has a relatively low performance of *T*-engine, due to its excessive CPU utilization and network consumption for frequent synchronization.

More specifically, in Fig. 14, we explore the variations in resource utilizations of the HTAP systems under their maximum supported *A*-threads as TPS changes. PT (**P**eak **T**PS) is the highest OLTP performance for each system under the default configuration, gradually reduced by 500 TPS for four groups of tests. PT is 6.5K and *A*-thread is 160 for PostgreSQL-SR, PT is 5K and *A*-thread is 130 for OceanBase, and PT is 2K and *A*-thread is 10 for TiDB. Except PostgreSQL-SR, which employs separate write (W) and read (R) nodes, the other two systems utilize the resources in a single node for both write and read. Therefore, we show PostgreSQL-SR's resource utilization for W and R separately in Fig. 14, that is PostgreSQL-SR(W)/(R). For all systems, we find that resource contention (especially in CPU) plays a significant role in bottleneck performance in scenarios with lower TPS, such as PT-1500 and PT-1000 for each system. Except PostgreSQL-SR(W), the CPU usages reach nearly 70-80% utilization for the large consumption of OLAP tasks. PostgreSQL-SR(W) with a low TPS consumes low CPU resource. However, for the high throughput scenarios (represented by PT), resources are not fully utilized, but we still cannot add more *A*-threads. The main reason is that the accumulation of data version chains of MVCC from the large writes and the garbage version holding by the long-running OLAP queries slow down new data write speed [20, 21]. Specifically, we illustrate the variation in execution latency for each type of TPC-C transaction under TPS=PT as the number of *A*-threads increases by a uniform fixed step from 1 to the maximum supported *A*-threads in Fig. 15. We collect the latency within each 1-min interval. Note that, most write transactions, e.g. *Delivery* and *Receive-Goods*, exhibit an increase in write latency even when resources are not fully utilized. Since PostgreSQL-SR takes a storage isolation method, it has little performance interference in the hybrid workloads.

But on PostgreSQL-SR(R), we observe a high number of dirty pages (2.2E+06 dirty pages) under its maximum supported *A*-threads. This is primarily due to the increased number of *A*-threads on R-Node keep significant historical data versions in the local cache; concurrently, its master write node introduces many new data for synchronization to the slave read node. Therefore, historical versions are not promptly flushed to disk, leading to the accumulation of dirty pages in the cache [39]. Meantime, PostgreSQL-SR needs to synchronize transaction logs from master write node to two slave nodes, which consume much network. But it offers relatively better isolation, and its resources (especially CPU and memory) are not fully utilized, even though it provides the best transaction performance. For OceanBase, compared to network and memory, CPU resources are consumed more, which is close to 80% utilization, since it can schedule extra CPU for OLAP queries. For TiDB, even though it deploys TiKV and TiFlash separately, it needs to confirm the consistency of data versions between *T*- and *A*-engines through frequent network communication. Therefore, it consumes the most network resources (5E+05 kbps).

In summary, different HTAP systems have different resource consumption patterns. Adaptively scheduling resources for *T*- and *A*-engines based on workload characteristics and system resource status, optimizing dirty page clean or data flush to disk, or retrofitting MVCC in version garbage collection and version chain searches, are worth more effort to improve overall performance.

6.2.2 Data Sharing. We control the workload interaction pattern by selecting ξ from 1, 3, 5 seconds(s) and β from 10%, 50%, 100% to simulate data sharing pressure under the default TPS=2K. In such a case, about 8K rows of new data can be generated per second. We select four common application scenarios with varying freshness requirements, i.e., freshness lag time $\mathcal{T}$, to assess the impact of data sharing models. These scenarios include fraud detection ($\mathcal{T}$=1-10ms), online gaming ($\mathcal{T}$=50-100ms), IOT system monitoring ($\mathcal{T}$=<200ms), and online e-commerce ($\mathcal{T}$=200-10000ms) [47]. We run one hundred of *FreshnessCheck* queries randomly while running *Vodka* for 10min and the results are summarized in Table 5.



(a) PostgreSQL-SR(0s) (b) PostgreSQL-SR(5s) (c) OceanBase(0s) (d) OceanBase(5s) (e) TiDB(0s) (f) TiDB(5s)

Figure 17: Unified Metric for Query FreshnessCheck

Table 5: Freshness(q) and Freshness Fail Rate in Various Business Scenarios Under Low TPS Scenarios

Systems	freshness(q)			freshness fail rate($\mathcal{T}$)			
	Min	Avg	Max	Fraud Detection[1]	Online Gaming[2]	IOT System Monitor[3]	Online E-Commerce[4]
PostgreSQL-SR(0s)	0ms	0ms	0ms	0%	0%	0%	0%
PostgreSQL-SR(5s)	4864ms	4988ms	5001ms	100%	100%	100%	0%
OceanBase(0s)	0ms	0ms	0ms	0%	0%	0%	0%
OceanBase(5s)	1431ms	1624ms	2526ms	100%	100%	100%	0%
TiDB(0s)	0ms	0ms	0ms	0%	0%	0%	0%
TiDB(5s)	583ms	753ms	1038ms	100%	100%	100%	0%

[1] **[1-10] ms.** [2] **[50-100] ms.** [3] **< 200 ms.** [4] **[200-10000] ms.**

PostgreSQL-SR provides an asynchronous synchronization. Its freshness relies on a pre-assigned synchronization interval by the parameter *'recovery_min_apply_delay'*. Its default value is 0, which means to restore transaction logs from master (write) node as soon as possible to the other two nodes. Though OceanBase and TiDB provides linear consistency, they also offer a weak consistency read mode and allow queries to read stale data within a specified *time lag range*. Since the minimum lag time threshold for OceanBase is 5s, it is selected for our experiments. OceanBase does not distinguish T-engine from A-engine strictly. It uses one copy of data serving hybrid workloads without data synchronization. TiDB achieves freshness through real-time data synchronization from TiKV to TiFlash using the multi-raft protocol. Both of them provide linear consistency, achieving $freshness(q)$=0 and *freshness fail rate* = 0 as shown in Table 5. However under a low TPS, PostgreSQL-SR exposes the same freshness as OceanBase and TiDB by setting its synchronization parameter to 0, i.e., PostgreSQL-SR(0s). So all of them satisfy the freshness requirements from the four applications. However, if we relax the read consistency of OceanBase and TiDB or synchronization time of PostgreSQL-SR to 5s, all three systems can no longer support (near) real-time business requirements but only businesses like E-Commerce. Among PostgreSQL (5s), OceanBase (5s) and TiDB (5s), TiDB (5s) has the best $freshness(q)$, benefiting from its optimization of consistency protocol, and PostgreSQL (5s) is the worst and just launches synchronization periodically, about every 5s. However, weak consistency read can improve an overall performance by relaxing the synchronization pressure (details in Technical Report [54]). Therefore, we can set the lag time according to user-expected freshness for a better performance.

Furthermore, we demonstrate the *unified linearly-consistent latency* ($Latency^u$) in Fig. 17, which measures the process ability by both the fresh data synchronization time, i.e., $freshness(q)$, and the query processing time, i.e., $latency(q)$, defined in § 4.3. We collect $Latency^u(q)$ for the real-time query $FreshnessCheck$. For PostgreSQL-SR (0s), its overall latency remains relatively low as we increase the number of accessed warehouses β and freshness data bound ξ. This is mainly due to its effective log synchronization of a small data size and log replaying mechanisms, as well as its bitmap index scan in a single node. However, when we set a larger synchronization time interval 5s, i.e., PostgreSQL-SR (5s), its $latency^u$ is dominated by data synchronization interval (about 5s). For OceanBase and TiDB, when we increase β and ξ, there is also a noticeable increase of $latency^u$. Performance of OceanBase (0s) is dominated by its processing latency instead of the delay of data synchronization from T-engine to A-engine. Its B+tree index allows for a quick data location, resulting in a lower $latency^u$. Similar to PostgreSQL-SR (5s), $latency^u$ of OceanBase (5s) is also dominated by its freshness lag time, but it is still much better than PostgreSQL-SR (5s). TiDB has an expensive distributed process cost, so compared to the other two HTAP systems, its $latency(q)$ dominates the performance for both TiDB (0s) and TiDB (5s).

Table 6: Freshness Comparison in PostgreSQL-SR and OceanBase under High TPS Scenarios

Freshness/Systems	PostgreSQL-SR	OceanBase
Min	0ms	0ms
Average	125ms	0ms
Max	504ms	0ms

Note that in order to make the results in three HTAP systems comparable, we have set TPS=2K. Since a low TPS generates a slow increase of data, a small data synchronization in PostgreSQL-SR has no impact on performance. We then increase TPS=5K and compare the performance between PostgreSQL-SR and OceanBase to investigate whether the data synchronization mechanism in PostgreSQL-SR is sufficient to meet actual HTAP requirements. To maximize the data replication, we set access ratio β = 100% and the fresh data bound ξ = 5s. We run 100 rounds of $FreshnessCheck$ and collect the information of $freshness(q)$ in Table 6. We observe that OceanBase always achieves the highest freshness with *freshness fail rate*=0; PostgreSQL-SR, under such a high TPS, has an average freshness latency of approximately 125ms and a maximum latency of 504ms, which does not meet the requirements for (near) real-time application scenarios such as fraud detection, online gaming, and IOT system monitor.

6.3 Experiment Summarization

After running $Vodka$ on HTAP database systems, we summarize the following three key findings.

1. To *resource isolation*, a complete storage isolation can yield optimal isolation effects but may result in significant resource waste on T-engine. In contrast, logical isolation is less effective, but it allows for a better resource utilization by unified resource management or adaptive scheduling strategies. A hybrid isolation solution is expected as mentioned in work [45], which prioritizes storage isolation, and then logical isolation when lack of resources.

2. To *data sharing*, under a mild OLTP throughput, an asynchronous network communication-based data sharing is comparable to an MVCC-based data sharing which needs no data migration for hybrid workloads. An HTAP system with the strict linear consistency model may be more applicable by providing an extra weak consistency read mode [34].

3. For HTAP systems, the capability of T- and A-engines has a more significant impact on the overall performance compared to implementation or optimization of *resource isolation* and *data sharing*. For example, on A-engine, vectorization and column-based storage are preferred; on T-engine, reducing distributed transactions is preferred. So an excellent HTAP system should first have high performance T- and A-engines.

7 CONCLUSION

This paper proposes a comprehensive benchmarking methodology for HTAP systems and implements a benchmark framework $Vodka$ on the basis of TPC-C and TPC-H. It is the first work to guarantee the benchmarking repeatability and fairness by 1) controlling the computational complexity as data changing, 2) designing an accurate performance model based on the data size and query latency, and 3) defining a unified metric to consolidate the impact of data sharing model and the query processing ability. $Vodka$ enables benchmarking HTAP systems in a comparable way for various resource isolation mechanisms and data sharing models.

REFERENCES

[1] Charu C. Aggarwal. 2006. On Biased Reservoir Sampling in the Presence of Stream Evolution. In Proceedings of the 32nd International Conference on Very Large Data Bases, Seoul, Korea, September 12-15, 2006, Umeshwar Dayal, Kyu-Young Whang, David B. Lomet, Gustavo Alonso, Guy M. Lohman, Martin L. Kersten, Sang Kyun Cha, and Young-Kuk Kim (Eds.). ACM, Seoul, Korea, 607–618. http://dl.acm.org/citation.cfm?id=1164180

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

[3] Altan Birler. 2019. Scalable reservoir sampling on many-core CPUs. In Proceedings of the 2019 International Conference on Management of Data. ACM New York, NY, USA, Amsterdam, The Netherlands, 1817–1819.

[4] Mokrane Bouzeghoub. 2004. A framework for analysis of data freshness. In Proceedings of the 2004 international workshop on Information quality in information systems. Association for Computing MachineryNew YorkNYUnited States, Paris France, 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. Proceedings of the VLDB Endowment 11, 12 (2018), 1849–1862.

[6] Jacopo Cavazza and Vittorio Murino. 2016. Active Regression with Adaptive Huber Loss. CoRR abs/1606.01568 (2016). arXiv:1606.01568 http://arxiv.org/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. Proceedings of the VLDB Endowment 15, 12 (2022), 3411–3424.

[8] Byonggon Chun, Jihun Ha, Sewon Oh, Hyunsung Cho, and MyeongGi Jeong. 2019. Kubernetes enhancement for 5G NFV infrastructure. In 2019 International Conference on Information and Communication Technology Convergence (ICTC). IEEE, IEEE, Jeju Island, Korea, 1327–1329.

[9] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. 2012. LazyBase: trading freshness for performance in a scalable database. In Proceedings of the 7th ACM european conference on Computer Systems. Association for Computing MachineryNew YorkNYUnited States, Bern Switzerland, 169–182.

[10] Fábio Coelho, João Paulo, Ricardo Vilaça, José Pereira, and Rui Oliveira. 2017. Htapbench: Hybrid transactional and analytical processing benchmark. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. Association for Computing MachineryNew YorkNYUnited States, L'Aquila Italy, 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 Proceedings of the Fourth International Workshop on Testing Database Systems. Association for Computing MachineryNew YorkNYUnited States, Athens Greece, 1–6.

[12] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2012. SAP HANA database: data management for modern business applications. ACM Sigmod Record 40, 4 (2012), 45–51.

[13] Gartner. 2014. Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation. https://www.gartner.com/en/documents/2657815.

[14] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. 2010. Hyrise: a main memory hybrid storage engine. Proceedings of the VLDB Endowment 4, 2 (2010), 105–116.

[15] Wassily Hoeffding. 1994. Probability inequalities for sums of bounded random variables. In The collected works of Wassily Hoeffding. Springer, America, 409–426.

[16] Paul W Holland and Roy E Welsch. 1977. Robust regression using iteratively reweighted least-squares. Communications in Statistics-theory and Methods 6, 9 (1977), 813–827.

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

[18] 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 2022 IEEE 38th International Conference on Data Engineering (ICDE). IEEE, (Virtual) Kuala Lumpur, Malaysia, 1822–1834. https://doi.org/10.1109/ICDE53745.2022.00182

[19] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In 2011 IEEE 27th International Conference on Data Engineering. IEEE, IEEE Computer Society1730 Massachusetts Ave., NW Washington, DCUnited States, none, 195–206.

[20] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived transactions made less harmful. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Portland OR USA, 495–510.

[21] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In Proceedings of the 2022 International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Philadelphia PA USA, 49–64.

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

[23] Alex Lambert, Dimitri Bouche, Zoltan Szabo, and Florence d'Alché Buc. 2022. Functional Output Regression with Infimal Convolution: Exploring the Huber and ϵ-insensitive Losses. In International Conference on Machine Learning. PMLR, PMLR, Baltimore, Maryland USA, 11844–11867.

[24] Sophie Lambert-Lacroix and Laurent Zwald. 2011. Robust regression through the Huber's criterion and adaptive lasso penalty. (2011).

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

[26] Guoliang Li and Chao Zhang. 2022. HTAP Databases: What is New and What is Next. In Proceedings of the 2022 International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Philadelphia PA USA, 2483–2488.

[27] Yuming Li, Rong Zhang, Xiaoyan Yang, Zhenjie Zhang, and Aoying Zhou. 2018. Touchstone: Generating Enormous Query-Aware Test Databases. In 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA, 575–586. https://www.usenix.org/conference/atc18/presentation/li-yuming

[28] Eric Lo, Nick Cheng, Wilfred W. K. Lin, Wing-Kai Hon, and Byron Choi. 2014. MyBenchmark: generating databases for query workloads. VLDB J. 23, 6 (2014), 895–913.

[29] 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 Proceedings of the 2021 International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Virtual Event China, 2530–2542.

[30] Darko Makreshanski, Jana Giceva, Claude Barthels, and Gustavo Alonso. 2017. BatchDB: Efficient isolated execution of hybrid OLTP+ OLAP workloads for interactive applications. In Proceedings of the 2017 ACM International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Chicago Illinois USA, 37–50.

[31] Elena Milkai, Yannis Chronis, Kevin P Gaffney, Zhihan Guo, Jignesh M Patel, and Xiangyao Yu. 2022. How Good is My HTAP System?. In Proceedings of the 2022 International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Philadelphia PA USA, 1810–1824.

[32] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data. Association for Computing

MachineryNew YorkNYUnited States, Vancouver Canada, 499–510.

[33] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast database joins and PSI for secret shared data. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. Association for Computing MachineryNew YorkNYUnited States, Virtual Event USA, 1271–1287.

[34] OceanBase. 2023. Weak consistency read. https://en.oceanbase.com/docs/common-oceanbase-database-10000000000870504.

[35] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). Pat 200, 0 (2007), 50.

[36] Rudolf Ernst Peierls. 1952. The commutation laws of relativistic field theory. Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences 214, 1117 (1952), 143–157.

[37] PingCAP. 2021. Empower Your Business with Big Data + Real-time Analytics in TiDB. https://medium.com/nerd-for-tech/empower-your-business-with-big-data-real-time-analytics-in-tidb-61e12645939b..

[38] PingCAP. 2023. TiFlash Overview. https://docs.pingcap.com/tidb/v7.2/tiflash-overview..

[39] PostgreSQL. 2018. High availability and scalable reads in PostgreSQL. https://www.timescale.com/blog/scalable-postgresql-high-availability-read-scalability-streaming-replication-fb95023e2af/.

[40] PostgreSQL. 2021. Swarm64 Benchmark. https://github.com/swarm64/s64da-benchmark-toolkit.

[41] PostgreSQL. 2023. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.

[42] PostgreSQL. 2023. Streaming Replication Protocol. https://www.postgresql.org/docs/current/protocol-replication.html.

[43] Luyi Qu, Yuming Li, Rong Zhang, Ting Chen, Ke Shu, Weining Qian, and Aoying Zhou. 2022. Application-Oriented Workload Generation for Transactional Database Performance Evaluation. In ICDE. IEEE, (Virtual) Kuala Lumpur, Malaysia, 420–432.

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

[45] Aunn Raza, Periklis Chrysogelos, Angelos Christos Anadiotis, and Anastasia Ailamaki. 2020. Adaptive HTAP through elastic resource scheduling. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Portland OR USA, 2043–2054.

[46] Anupam Sanghi, Raghav Sood, Jayant R. Haritsa, and Srikanta Tirthapura. 2018. Scalable and Dynamic Regeneration of Big Data Volumes. In EDBT. OpenProceedings.org, Vienna, Austria, 301–312.

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

[48] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance characterization of HTAP workloads. In 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, IEEE, Chania, Greece, 1829–1834.

[49] TiDB. 2023. Hybrid Deployment Topology. https://docs.pingcap.com/tidb/stable/hybrid-deployment-topology.

[50] TPC. 1999. TPC-DS Benchmark. http://www.tpc.org/tpcds/.

[51] TPC. 1999. TPC-H Benchmark. http://www.tpc.org/tpch/.

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

[53] 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. Proceedings of the VLDB Endowment 15, 10 (2022), 1991–2004.

[54] Vodka. 2023. Technical Report And Source Codes Of Vodka. https://github.com/Wind-Gone/Vodka-HTAP-Benchmark.

[55] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In Proceedings of the 2022 International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Philadelphia PA USA, 94–107.

[56] Wikipedia. 2023. Bayes' theorem. https://en.wikipedia.org/.

[57] Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. 2023. Communication-Optimal Parallel Reservoir Sampling. In BTW (LNI), Vol. P-331. Gesellschaft für Informatik e.V., 567–578.

[58] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An empirical evaluation of in-memory multi-version concurrency control. Proceedings of the VLDB Endowment 10, 7 (2017), 781–792.

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

[60] 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. Proceedings of the VLDB Endowment 16, 12 (2023), 3728–3740.

[61] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, et al. 2022. OceanBase: a 707 million tpmC distributed relational database system. Proceedings of the VLDB Endowment 15, 12 (2022), 3385–3397.

[62] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, et al. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. Proceedings of the VLDB Endowment 12, 12 (2019), 2059–2070.

[63] 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 Companion of the 2023 International Conference on Management of Data. Association for Computing MachineryNew YorkNYUnited States, Washington, USA, 95–98.

A TABLE SCHEMA

The schema in *Vodka* is shown in Fig. 18.

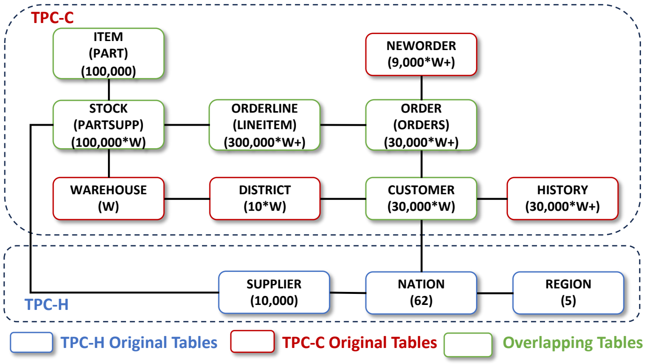


Figure 18: Table Schema in *Vodka*

B WORKLOADS

The transactions and query templates are listed as follows.

B.1 Transactions

New Order

```sql
SELECT c_discount, c_last, c_credit, w_tax
FROM vodka_customer
JOIN vodka_warehouse ON (w_id = c_w_id)
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;

SELECT d_tax, d_next_o_id
FROM vodka_district
WHERE d_w_id = ? AND d_id = ?
FOR UPDATE;

UPDATE vodka_district
SET d_next_o_id = d_next_o_id + 1
WHERE d_w_id = ? AND d_id = ?;

INSERT INTO vodka_oorder (
    o_id, o_d_id, o_w_id, o_c_id, o_entry_d,
    o_ol_cnt, o_all_local, o_shippriority
)
VALUES (?, ?, ?, ?, ?, ?, ?, ?);

INSERT INTO vodka_new_order (
    no_o_id, no_d_id, no_w_id
)
VALUES (?, ?, ?);

SELECT s_quantity, s_data,
       s_dist_01, s_dist_02, s_dist_03, s_dist_04,
       s_dist_05, s_dist_06, s_dist_07, s_dist_08,
       s_dist_09, s_dist_10
FROM vodka_stock
WHERE s_w_id = ? AND s_i_id = ?
FOR UPDATE;

SELECT i_price, i_name, i_data
FROM vodka_item
WHERE i_id = ?;

UPDATE vodka_stock
SET s_quantity = ?, s_ytd = s_ytd + ?,
    s_order_cnt = s_order_cnt + 1,
    s_remote_cnt = s_remote_cnt + ?
WHERE s_w_id = ? AND s_i_id = ?;

INSERT INTO vodka_order_line (
    ol_o_id, ol_d_id, ol_w_id, ol_number,
    ol_i_id, ol_supply_w_id, ol_quantity,
    ol_amount, ol_dist_info, ol_discount, ol_commitdate,
         ol_suppkey, access_version
)
VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
```

Delivery

```sql
SELECT no_o_id
FROM vodka_new_order
WHERE no_w_id = ? AND no_d_id = ?
ORDER BY no_o_id ASC;

DELETE FROM vodka_new_order
WHERE no_w_id = ? AND no_d_id = ? AND no_o_id = ?;

SELECT o_c_id, o_entry_d
FROM vodka_oorder
WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?;

SET o_carrier_id = ?
WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?;

SELECT sum(ol_amount) AS sum_ol_amount
FROM vodka_order_line
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?;

SELECT ol_delivery_d
FROM vodka_order_line
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?;

UPDATE vodka_order_line
SET ol_delivery_d = ?, ol_shipmode = ?, ol_shipinstruct =
      ?, access_version = 1
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?;

UPDATE vodka_customer
SET c_balance = c_balance + ?, c_delivery_cnt =
      c_delivery_cnt + 1
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;
```

Order Status

```sql
SELECT c_id
FROM vodka_customer
WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?
ORDER BY c_first;

SELECT c_first, c_middle, c_last, c_balance
FROM vodka_customer
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;

SELECT o_id, o_entry_d, o_carrier_id
FROM vodka_oorder
WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
  AND o_id = (
      SELECT max(o_id)
      FROM vodka_oorder
      WHERE o_w_id = ? AND o_d_id = ? AND o_c_id = ?
  );

SELECT ol_i_id, ol_supply_w_id, ol_quantity,
       ol_amount, ol_delivery_d
FROM vodka_order_line
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?
ORDER BY ol_w_id, ol_d_id, ol_o_id, ol_number;
```

Payment

```sql
SELECT w_name, w_street_1, w_street_2, w_city,
       w_state, w_zip
FROM vodka_warehouse
WHERE w_id = ?;

SELECT d_name, d_street_1, d_street_2, d_city,
       d_state, d_zip
FROM vodka_district
WHERE d_w_id = ? AND d_id = ?;

SELECT c_id
FROM vodka_customer
WHERE c_w_id = ? AND c_d_id = ? AND c_last = ?
ORDER BY c_first;

SELECT c_first, c_middle, c_last, c_street_1, c_street_2,
      c_city, c_nationkey, c_zip, c_phone, c_since, c_credit,
      c_credit_lim, c_discount, c_balance
FROM vodka_customer
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?
FOR UPDATE;

SELECT c_data
FROM vodka_customer
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;

UPDATE vodka_warehouse
SET w_ytd = w_ytd + ?
WHERE w_id = ?;

UPDATE vodka_district
SET d_ytd = d_ytd + ?
WHERE d_w_id = ? AND d_id = ?;

UPDATE vodka_customer
SET c_balance = c_balance - ?,
    c_ytd_payment = c_ytd_payment + ?,
    c_payment_cnt = c_payment_cnt + 1
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;

UPDATE vodka_customer
SET c_balance = c_balance - ?,
    c_ytd_payment = c_ytd_payment + ?,
    c_payment_cnt = c_payment_cnt + 1,
    c_data = ?
WHERE c_w_id = ? AND c_d_id = ? AND c_id = ?;

INSERT INTO vodka_history (
    h_c_id, h_c_d_id, h_c_w_id, h_d_id, h_w_id,
    h_date, h_amount, h_data
)
VALUES (?, ?, ?, ?, ?, ?, ?, ?);
```

Receive Goods

```
SELECT ol_delivery_d
FROM vodka_order_line
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?;

UPDATE vodka_order_line
SET ol_receipdate = ?, ol_returnflag = ?
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?;

UPDATE vodka_oorder
SET o_comment = ?
WHERE o_w_id = ? AND o_d_id = ? AND o_id = ?;

SELECT ol_receipdate
FROM vodka_order_line
WHERE ol_w_id = ? AND ol_d_id = ? AND ol_o_id = ?;
```

```
SELECT count(*) AS low_stock
FROM (
    SELECT s_w_id, s_i_id, s_quantity
    FROM vodka_stock
    WHERE s_w_id = ? AND s_quantity < ?
    AND s_i_id IN (
        SELECT ol_i_id
        FROM vodka_district
        JOIN vodka_order_line ON ol_w_id = d_w_id
                          AND ol_d_id = d_id
                          AND ol_o_id >= d_next_o_id - 20
                          AND ol_o_id < d_next_o_id
        WHERE d_w_id = ? AND d_id = ?
    )
) AS L;
```

B.2 Queries

Q1

```
    SELECT
    ol_number,
    ol_returnflag,
    sum(ol_quantity) as sum_qty,
    sum(ol_amount) as sum_base_price,
    sum(ol_amount * (1 - ol_discount)) as sum_disc_price,
    sum(ol_amount * (1 - ol_discount) * (1 + ol_tax)) as sum_charge,
    avg(ol_quantity) as avg_qty,
    avg(ol_amount) as avg_price,
    avg(ol_discount) as avg_disc,
    count(*) as count_order
FROM vodka_order_line
WHERE ol_delivery_d <= date '1998-12-01'
GROUP BY ol_returnflag, ol_number
ORDER BY ol_returnflag, ol_number;
```

Q2

```
    SELECT
    s_acctbal,
    s_name,
    n_name,
    i_id,
    i_mfgr,
    s_address,
    s_phone,
    s_comment
FROM
    vodka_item,
    vodka_supplier,
    vodka_stock,
    vodka_nation,
    vodka_region
WHERE
    i_id = s_i_id
    AND s_suppkey = s_tocksuppkey
    AND i_size = 15
    AND i_type like '%BRASS'
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'EUROPE'
    AND s_supplycost = (
        SELECT
            min(s_supplycost)
        FROM
            vodka_stock,
            vodka_supplier,
            vodka_nation,
            vodka_region
        WHERE
            i_id = s_i_id
            AND s_suppkey = s_stocksuppkey
            AND s_nationkey = n_nationkey
            AND n_regionkey = r_regionkey
            AND r_name = 'EUROPE'
    )
ORDER BY
    s_acctbal DESC,
    n_name,
    s_name,
    i_id
LIMIT 100;
```

Q3

```
    SELECT
    ol_w_id, ol_d_id, ol_o_id,
    sum(ol_amount * (1 - ol_discount)) as revenue,
    o_entry_d,
    o_shippriority
FROM
    vodka_customer,
    vodka_order_line,
    vodka_oorder
WHERE
    c_mktsegment = 'BUILDING'
    AND c_w_id = o_w_id
    AND c_d_id = o_d_id
    AND c_id = o_c_id
    AND ol_w_id = o_w_id
    AND ol_d_id = o_d_id
    AND ol_o_id = o_id
    AND ol_delivery_d > date '1998-12-01'
GROUP BY
    ol_w_id, ol_d_id, ol_o_id,
    o_entry_d,
    o_shippriority
ORDER BY
    revenue DESC,
    o_entry_d
LIMIT 10;
```

Q4

```
    SELECT
    o_carrier_id,
    count(*) as order_count
FROM
    vodka_oorder
WHERE
    o_entry_d < date '1998-12-01'
    AND o_entry_d >= date '1998-09-01'
    AND EXISTS (
        SELECT
            *
        FROM
            vodka_order_line
        WHERE
            ol_w_id = o_w_id
            AND ol_d_id = o_d_id
            AND ol_o_id = o_id
            AND ol_commitdate < ol_receipdate
    )
GROUP BY
    o_carrier_id
ORDER BY
    o_carrier_id;
```

Q5

```
    SELECT
    n_name,
    sum(ol_amount * (1 - ol_discount)) as revenue
FROM
    vodka_customer,
    vodka_oorder,
    vodka_order_line,
    vodka_supplier,
    vodka_nation,
    vodka_region
WHERE
    c_w_id = o_w_id and c_d_id = o_d_id and c_id = o_c_id
    AND ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id
         = o_id
    AND ol_suppkey = s_suppkey
    AND c_nationkey = s_nationkey
    AND s_nationkey = n_nationkey
    AND n_regionkey = r_regionkey
    AND r_name = 'ASIA'
    AND o_entry_d < date '1998-12-01'
    AND o_entry_d >= date '1997-12-01'
GROUP BY
    n_name
ORDER BY
    revenue DESC;
```

Q6

```
    SELECT
    sum(ol_amount * ol_discount) as revenue
FROM
    vodka_order_line
WHERE
    ol_delivery_d < date '1998-12-01'
    AND ol_delivery_d >= date '1997-12-01'
    AND ol_discount BETWEEN 0.05 AND 0.07
    AND ol_quantity < 24;
```

Q7

```
    SELECT
    supp_nation,
    cust_nation,
    l_year,
    sum(volume) as revenue
FROM
    (
        SELECT
            n1.n_name as supp_nation,
            n2.n_name as cust_nation,
            extract(year from ol_delivery_d) as l_year,
            ol_amount * (1 - ol_discount) as volume
        FROM
            vodka_supplier,
            vodka_order_line,
            vodka_oorder,
            vodka_customer,
            vodka_nation n1,
            vodka_nation n2
        WHERE
            s_suppkey = ol_suppkey
            AND ol_w_id = o_w_id
            AND ol_d_id = o_d_id
            AND ol_o_id = o_id
            AND c_w_id = o_w_id
            AND c_d_id = o_d_id
            AND c_id = o_c_id
            AND s_nationkey = n1.n_nationkey
            AND c_nationkey = n2.n_nationkey
            AND (
                (n1.n_name = 'FRANCE' AND n2.n_name = 'GERMANY')
                OR (n1.n_name = 'GERMANY' AND n2.n_name = 'FRANCE')
            )
            AND ol_delivery_d < date '1998-12-01'
            AND ol_delivery_d >= date '1996-12-01'
    ) as shipping
GROUP BY
    supp_nation,
    cust_nation,
    l_year
ORDER BY
    supp_nation,
    cust_nation,
    l_year;
```

Q8

```
    SELECT
    o_year,
    sum(CASE
        WHEN nation = 'BRAZIL' THEN volume
        ELSE 0
    END) / (sum(volume) + 0.001) as mkt_share
FROM
    (
        SELECT
            extract(year from o_entry_d) as o_year,
            ol_amount * (1 - ol_discount) as volume,
            n2.n_name as nation
        FROM
            vodka_item,
            vodka_supplier,
            vodka_order_line,
            vodka_oorder,
            vodka_customer,
            vodka_nation n1,
            vodka_nation n2,
            vodka_region
        WHERE
            i_id = ol_i_id
            AND s_suppkey = ol_suppkey
            AND ol_w_id = o_w_id
            AND ol_d_id = o_d_id
            AND ol_o_id = o_id
            AND c_w_id = o_w_id
            AND c_d_id = o_d_id
            AND c_id = o_c_id
            AND c_nationkey = n1.n_nationkey
            AND n1.n_regionkey = r_regionkey
            AND r_name = 'AMERICA'
            AND s_nationkey = n2.n_nationkey
            AND o_entry_d < date '1998-12-01'
            AND o_entry_d >= date '1996-12-01'
            AND i_type = 'ECONOMY ANODIZED STEEL'
    ) as all_nations
GROUP BY
    o_year
ORDER BY
    o_year;
```

Q9

```
    SELECT
    nation,
    o_year,
    sum(amount) as sum_profit
FROM
    (
        SELECT
            n_name as nation,
            extract(year from o_entry_d) as o_year,
            ol_amount * (1 - ol_discount) - s_supplycost * ol_quantity as amount
        FROM
            vodka_item,
            vodka_supplier,
            vodka_order_line,
            vodka_stock,
            vodka_oorder,
            vodka_nation
        WHERE
            s_suppkey = ol_suppkey
            AND s_tocksuppkey = ol_suppkey
            AND s_i_id = ol_i_id
            AND i_id = ol_i_id
            AND o_w_id = ol_w_id
            AND o_d_id = ol_d_id
            AND o_id = ol_o_id
            AND s_nationkey = n_nationkey
            AND i_name like '%green%'
    ) as profit
GROUP BY
    nation,
    o_year
ORDER BY
    nation,
    o_year desc;
```

Q10

```
    SELECT
    c_w_id, c_d_id, c_id,
    c_last,
    sum(ol_amount * (1 - ol_discount)) as revenue,
    c_balance,
    n_name,
    c_phone
FROM
    vodka_customer,
    vodka_oorder,
    vodka_order_line,
    vodka_nation
WHERE
    c_w_id = o_w_id and c_d_id = o_d_id and c_id = o_c_id
    and ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id
         = o_id
    and o_entry_d < date '1995-03-15'
    and o_entry_d >= date '1995-03-15' - interval '3'
         month
    and ol_returnflag = 'R'
    and c_nationkey = n_nationkey
GROUP BY
    c_w_id, c_d_id, c_id,
    c_last,
    c_balance,
    c_phone,
    n_name
ORDER BY
    revenue desc limit 20;
```

Q11

```
    SELECT
    s_i_id,
    sum(s_supplycost * s_quantity) as value
FROM
    vodka_stock,
    vodka_supplier,
    vodka_nation
WHERE
    s_tocksuppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = 'GERMANY'
GROUP BY
    s_i_id
HAVING
    sum(s_supplycost * s_quantity) > (
        select
        sum(s_supplycost * s_quantity) * 0.01
        from
        vodka_stock,
        vodka_supplier,
        vodka_nation
        where
        s_tocksuppkey = s_suppkey
        and s_nationkey = n_nationkey
        and n_name = 'GERMANY'
    )
ORDER BY
    value desc;
```

Q12

```
    SELECT
    ol_shipmode,
    sum(case
        when o_carrier_id = 1
            or o_carrier_id = 2
            then 1
        else 0
    end) as high_line_count,
    sum(case
        when o_carrier_id <> 1
            and o_carrier_id <> 2
            then 1
        else 0
    end) as low_line_count
FROM
    vodka_oorder,
    vodka_order_line
WHERE
    ol_w_id = o_w_id
    and ol_d_id = o_d_id
    and ol_o_id = o_id
    and ol_shipmode in ('MAIL', 'SHIP')
    and ol_commitdate < ol_receipdate
    and ol_delivery_d < ol_commitdate
    and ol_receipdate < date '1995-01-01'
    and ol_receipdate >= date '1995-01-01' - interval '1'
         year
GROUP BY
    ol_shipmode
```

```
ORDER BY
    ol_shipmode;
```

Q13

```
    select
    c_count,
    count(*) as custdist
from
    (
        select
            c_w_id, c_d_id, c_id,
            count(*)
        from
            vodka_customer left outer join vodka_oorder on
                       c_w_id = o_w_id and c_d_id = o_d_id
                            and c_id = o_c_id
                   and o_comment not like
                        '%special%requests%'
        group by
            c_w_id, c_d_id, c_id
    ) as c_orders (c_custkey, c_count)
group by
    c_count
order by
    custdist desc,
    c_count desc;
```

Q14

```
    select
    100.00 * sum(case
                    when i_type like 'PROMO%'
                        then ol_amount * (1 - ol_discount)
                    else 0
                end) / (sum(ol_amount * (1 -
                     ol_discount))+0.001) as promo_revenue
from
    vodka_order_line,
    vodka_item
where
        ol_i_id = i_id
  and ol_delivery_d < date '1995-03-15'
  and ol_delivery_d >= date '1995-02-15';
```

Q15

```
    with revenue (supplier_no, total_revenue) as
(
    select ol_suppkey,
           sum(ol_amount * (1 - ol_discount))
    from vodka_order_line
    where
            ol_delivery_d < TIMESTAMP '1995-03-15'
        and ol_delivery_d >= TIMESTAMP '1995-02-15' -
             interval '3' month
    group by ol_suppkey
)
select s_suppkey,
       s_name,
       s_address,
       s_phone,
       total_revenue
from vodka_supplier, revenue
where s_suppkey = supplier_no
  and total_revenue = (select max(total_revenue)
                       from revenue)
order by s_suppkey;
```

Q16

```
    select
    i_brand,
    i_type,
    i_size,
    count(distinct s_tocksuppkey) as supplier_cnt
from
    vodka_stock,
    vodka_item
where
    i_id = s_i_id
    and i_brand <> 'Brand#45'
    and i_type not like 'MEDIUM POLISHED%'
    and i_size in (49, 14, 23, 45, 19, 3, 36, 9)
    and s_tocksuppkey not in (
        select s_suppkey
        from vodka_supplier
        where
            s_comment like '%Customer%Complaints%'
            and s_suppkey < 500
    )
group by
    i_brand,
    i_type,
    i_size
order by
    supplier_cnt desc,
    i_brand,
    i_type,
    i_size;
```

Q17

```
    select
    sum(ol_amount) / 7.0 as avg_yearly
from
    vodka_order_line,
    vodka_item
where
    i_id = ol_i_id
    and i_brand = 'Brand#23'
    and i_container = 'MED BOX'
    and ol_quantity < (
        select
            0.2 * avg(ol_quantity)
        from
            vodka_order_line
        where
            ol_i_id = i_id
    );
```

Q18

```
    select
    c_last,
    c_w_id, c_d_id, c_id,
    o_w_id, o_d_id, o_id,
    o_entry_d,
    sum(ol_quantity) AS o_totalprice
from
    vodka_customer,
    vodka_oorder,
    vodka_order_line
where
    (o_w_id, o_d_id, o_id) in (
        select
            ol_w_id, ol_d_id, ol_o_id
        from
            vodka_order_line
        group by
            ol_w_id, ol_d_id, ol_o_id
        having
            sum(ol_quantity) > 300
    )
    and c_w_id = o_w_id and c_d_id = o_d_id and c_id = o_c_id
    and ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id = o_id
group by
    c_last,
    c_w_id, c_d_id, c_id,
    o_w_id, o_d_id, o_id,
    o_entry_d
order by
    o_totalprice DESC,
    o_entry_d
limit 100;
```

Q19

```
    select
    c_last,
    c_w_id, c_d_id, c_id,
    o_w_id, o_d_id, o_id,
    o_entry_d,
    sum(ol_quantity) AS o_totalprice
from
    vodka_customer,
    vodka_oorder,
    vodka_order_line
where
    (o_w_id, o_d_id, o_id) in (
        select
            ol_w_id, ol_d_id, ol_o_id
        from
            vodka_order_line
        group by
            ol_w_id, ol_d_id, ol_o_id
        having
            sum(ol_quantity) > 300
    )
    and c_w_id = o_w_id and c_d_id = o_d_id and c_id = o_c_id
    and ol_w_id = o_w_id and ol_d_id = o_d_id and ol_o_id = o_id
group by
    c_last,
    c_w_id, c_d_id, c_id,
    o_w_id, o_d_id, o_id,
    o_entry_d
order by
    o_totalprice DESC,
    o_entry_d
limit 100;
```

Q20

```
    select
    s_name,
    s_address
from
    vodka_supplier,
    vodka_nation
where
    s_suppkey in (
    select
        s_tocksuppkey
    from
        vodka_stock
    where
        s_i_id in (
        select
            i_id
        from
            vodka_item
        where
            i_name like 'forest%'
    )
      and s_quantity > (
        select
                0.5 * sum(ol_quantity)
        from
            vodka_order_line
        where
                ol_i_id = s_i_id
          and ol_suppkey = s_tocksuppkey
          and ol_delivery_d < TIMESTAMP '1995-01-01'
          and ol_delivery_d >= TIMESTAMP '1994-01-01' - interval '1' year
    )
  )
  and s_nationkey = n_nationkey
  and n_name = 'CANADA'
order by
    s_name;
```

Q21

```
    select
    s_name,
    count(*) as numwait
from
    vodka_supplier,
    vodka_order_line l1,
    vodka_oorder,
    vodka_nation
where
    s_suppkey = l1.ol_suppkey
  and l1.ol_w_id=o_w_id and l1.ol_d_id=o_d_id and
       l1.ol_o_id=o_id
  and l1.ol_receipdate > l1.ol_commitdate
  and exists (
    select
        *
    from
        vodka_order_line l2
    where
        l2.ol_w_id=l1.ol_w_id and l2.ol_d_id=l1.ol_w_id
             and l2.ol_o_id=l1.ol_o_id
      and l2.ol_suppkey <> l1.ol_suppkey
  )
  and not exists (
    select
        *
    from
        vodka_order_line l3
    where
        l3.ol_w_id=l1.ol_w_id and l3.ol_d_id=l1.ol_w_id
             and l3.ol_o_id=l1.ol_o_id
      and l3.ol_suppkey <> l1.ol_suppkey
      and l3.ol_receiptdate > l3.ol_commitdate
  )
  and s_nationkey = n_nationkey
  and n_name = 'SAUDI ARABIA'
group by
    s_name
order by
    numwait desc,
    s_name limit 100;
```

Q22

```
    select
    cntrycode,
    count(*) as numcust,
    sum(c_balance) as totacctbal
from
    (
        select
            substring(c_phone from 1 for 2) as cntrycode,
            c_balance
        from
            vodka_customer
        where
                substring(c_phone from 1 for 2) in
                ('13', '31', '23', '29', '30', '18', '17')
          and c_balance > (
            select
                avg(c_balance)
            from
                vodka_customer
            where
                    c_balance > 0.00
              and substring(c_phone from 1 for 2) in
                  ('13', '31', '23', '29', '30', '18', '17')
        )
          and not exists (
                select
                    *
                from
                    vodka_oorder
                where
                        c_w_id = o_w_id and c_d_id = o_d_id
                             and c_id = o_c_id
            )
    ) as custsale
group by
    cntrycode
order by
    cntrycode;
```

B.3 More Join Type Discussion

We next take the most complex **Case 4** as an example to demonstrate the linear changes of other join types besides equi-join in § 3. In theory, any join type can be represented by the linear combination of *equi-join* ($\bowtie_{P_J}$) and *left semi-join* ($\ltimes_{P_J}$), and we give the detailed deduction of *left semi-join* as follows since *equi-join* has been represented in the body of our paper. Therefore, each join type operator can keep linear change and maintain consistent data access distribution.

- **Equi Join** The cardinality of equi-join has been deduced as by $|(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} (R_j \cup \Delta\mathrm{R_j})| = (1+\frac{|\Delta\mathrm{R_j}|}{|R_j|})|R_i \bowtie_{P_J} R_j|$.
- **Left Semi Join** The cardinality of left semi join can be represented as Equ. 22.

$$
\begin{aligned}
&|(R_i \cup \Delta\mathrm{R_i}) \ltimes_{P_J} (R_j \cup \Delta\mathrm{R_j})| \\
&= (R_i \cup \Delta\mathrm{R_i}) \ltimes_{P_J} (R_j \cup \Delta\mathrm{R_j}) \\
&= |R_i \ltimes P_J R_j| + |(R_i \ltimes P_J \Delta\mathrm{R_j}{}^o) \cup (\Delta\mathrm{R_i} \ltimes_{P_J} \Delta\mathrm{R_j}{}^n)| \\
&= |R_i \ltimes_{P_J} R_j \cup (R_i \ltimes P_J \Delta\mathrm{R_j}{}^o)| + |(\Delta\mathrm{R_i} \ltimes P_J \Delta\mathrm{R_j}{}^n)| \\
&= |R_i \ltimes_{P_J} R_j| + |\Delta\mathrm{R_i} \ltimes P_J \Delta\mathrm{R_j}{}^n| \\
&= |R_i \ltimes_{P_J} R_j| + \frac{|\Delta\mathrm{R_i}|}{|R_i|}|R_i \ltimes P_J R_j| = (1+\frac{|\Delta\mathrm{R_i}|}{|R_i|})|R_i \ltimes_{P_J} R_j| \qquad (22)
\end{aligned}
$$

- **Right Semi Join** The right semi join is equivalent to the equi-join, represented as $|(R_i \cup \Delta\mathrm{R_i}) \rtimes_{P_J} (R_j \cup \Delta\mathrm{R_j})| = (1+\frac{|\Delta\mathrm{R_j}|}{|R_j|})|R_i \rtimes_{P_J} R_j|$.
- **Left Anti Join** The left anti join is equivalent to the size of left referenced relation minus the size of left semi join, which size can be represented by $|(R_i \cup \Delta\mathrm{R_i}) \triangleright_{P_J} (R_j \cup \Delta\mathrm{R_j})| = |(R_i \cup \Delta\mathrm{R_i})| - |(R_i \cup \Delta\mathrm{R_i}) \ltimes P_J(R_j \cup \Delta\mathrm{R_j})|$.
- **Right Anti Join** The right aanti join is equivalent to the size of right referencing relation minus the size of equi-join, which size can be represented by $|(R_i \cup \Delta\mathrm{R_i}) \triangleleft_{P_J} (R_j \cup \Delta\mathrm{R_j})| = |(R_j \cup \Delta\mathrm{R_j})| - |(R_i \cup \Delta\mathrm{R_i}) \bowtie P_J(R_j \cup \Delta\mathrm{R_j})|$
- **Left Outer Join** The left outer join is equivalent to the union of the left anti join and the equi-join, which size can be represented by $|(R_i \cup \Delta\mathrm{R_i}) \text{⟕}_{P_J} (R_j \cup \Delta\mathrm{R_j})| = |(R_i \cup \Delta\mathrm{R_i}) \triangleright_{P_J} (R_j \cup \Delta\mathrm{R_j})| + |(R_i \cup \Delta\mathrm{R_i}) \bowtie_{P_J} (R_j \cup \Delta\mathrm{R_j})|$
- **Right Outer Join** The right outer join is equivalent to the size of right referencing relation, which size can be represented by $|(R_i \cup \Delta\mathrm{R_i}) \text{⟖}_{P_J} (R_j \cup \Delta\mathrm{R_j})| = |R_j \cup \Delta\mathrm{R_j})|$.
- **Full Outer Join** The full outer join is equivalent to the union of the left join and the right referencing relation, which size can be represented by $|(R_i \cup \Delta\mathrm{R_i}) \text{⟗}_{P_J} (R_j \cup \Delta\mathrm{R_j})| = |(R_i \cup \Delta\mathrm{R_i}) \ltimes P_J(R_j \cup \Delta\mathrm{R_j})| + |R_j \cup \Delta\mathrm{R_j})|$