Vodka: Benchmarking HTAP Systems on Demand

Zirui Hu[1], Qingshuai Wang[1], Yao Luo[1], Jinkai Xu[1], Rong Yu[1], Rong Zhang[1], Lyu Ni[1], Xuan Zhou,[1]
Aoying Zhou[1], Quanqing Xu[2], Chuanhui Yang[2]
[1]East China Normal University
[2]OceanBase, Ant Group
Shanghai, China
{zrhu,qswang,yaoluo,jinkaixu,ryu}@stu.ecnu.edu.cn,{rzhang,lni,xzhou,ayzhou}@dase.ecnu.edu.cn
{xuquanqing.xqq,rizhao.ych}@oceanbase.com

ABSTRACT

In order to more fully exploit the value of fresh data, hybrid transaction/analytical processing(HTAP) systems that support fast access to the latest OLTP modifications within a single system for OLAP services analytics have become mainstream nowadays. Recent years witness the flourish of various HTAP systems and optimization strategies. Unfortunately, we still lack a benchmark that can fairly compare different systems in their key techniques of *resource isolation* and *data sharing*. In this paper, we seek to explore the genuine methodology of HTAP benchmark design starting from the HTAP key techniques. To fairly compare the capability of resource isolation reflected by performance of OLTP and OLAP, coupled with the performance interference degree between hybrid workloads, we introduce a new benchmarking method of controlling workloads based on *reservoir sampling* method to maintain a stable cardinality output and modeling query latency and data size based on *Robust Regression* to achieve a fair comparison of performance at any data size. By this, we can avoid the different complexity of the same OLAP workloads caused by differences in OLTP data expansion speeds and synchronization speeds. To capture the capability of data sharing, we propose a *query-oriented data freshness* definition, which focuses more on query accessing data range freshness level rather than merely system global freshness status. We further sketch hybrid workloads interaction patterns in controlling OLTP latest written range and OLAP read range to simulate different data sharing pressure and use *freshness fail rate* to demonstrate real-time analytics. Based on the above methodology, we design a new HTAP benchmark called *Vodka*. We conduct extensive experiments on three representative systems to demonstrate their performance and freshness under diverse application scenarios.

CCS CONCEPTS

• **Information systems → Database performance evaluation**.

KEYWORDS

HTAP, benchmark, cardinality control, query-oriented freshness, data consistency

ACM Reference Format:
Zirui Hu[1], Qingshuai Wang[1], Yao Luo[1], Jinkai Xu[1], Rong Yu[1], Rong Zhang[1], Lyu Ni[1], Xuan Zhou,[1], Aoying Zhou[1], Quanqing Xu[2], Chuanhui Yang[2]. 2023. Vodka: Benchmarking HTAP Systems on Demand. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 18 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

A Rising Trend in HTAP. In the domain of database systems (DBMS), online transaction processing (OLTP) is supposed to produce data, which can be used further for analysis; online analytical processing (OLAP) usually takes the static data that has been migrated by an ETL process for analysis. These two classic categories of systems have lived apart for a long time. To satisfy the increasing demand for real-time analytics, e.g., online exception detection/-monitoring [43] or accurate recommendation [45], and make full use of the data value [36], after a long separation of OLTP-engine (*T*-engine) and OLAP-engine (*A*-engine), we are starting a new journey of executing hybrid workloads in a class of new engines, which are called Hybrid Transactional and Analytical Processing Systems (HTAP systems) [16]. HTAP systems strive to accomplish business analytics directly over newly produced data.

New Technical Demands For HTAP. *T*-engine aims to enable a high throughput of small-sized transactions, which typically operates on a limited number of row-based tuples with a data schema of a reduced redundancy by normalization techniques across entities [14, 35]. But *A*-engine introduces a high redundancy in the shape of materialized views or precomputed aggregates, and operates on column-based data for improving query performance. It is obvious that *T*- and *A*-engines are not consistent in their implementations for either data management or workload processing. To alleviate the heterogeneity and guarantee performance of hybrid workloads, a massive flurry of giant database vendors participate in constructing HTAP systems [7, 9, 15, 20, 22, 24, 27, 58, 60]. Typically, we divide the implementation architectures into two types called *unified storage*, and *decoupled storage* based on the way of sharing storages [44, 53]. In unified storage HTAP systems, e.g., Oracle [24], SAP HANA [15] and OceanBase [59], they tend to execute hybrid workloads in a single node with the highest data freshness. To avoid/mitigate interferences from each other, they try to isolate OLTP and OLAP services by retrofitting traditional

Conference acronym 'XX, June 03–05, 2023, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

technology like using multiversion concurrency control (MVCC) and snapshotting mechanisms. Some of them also maintain dedicated in-memory column storage for OLAP and fetch logs of OLTP workloads to update column storage data and ensure freshness, such as Oracle [24]. The decoupled storage is also taken by many mainstream HTAP systems, like TiDB [20], F1 Lightning [58], and BatchDB [31], they separate T- and A-engines on different nodes with the purpose of a thorough physical resource isolation. Then we can import the fresh data produced by T-engine asynchronously into A-engine, which achieves a relatively high performance at the cost of data freshness. If it requires providing the freshness data on demand, a possible way is to check and import the fresh data version before launching OLAP processing as in TiDB, which inevitably incurs additional inter-node communication costs and lowers down the runtime performance of T-/A-engine. 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 so many HTAP systems and their optimizations, benchmarking has always been a fair way for horizontal comparison among systems. Traditionally, there has been extensive research on benchmarking OLTP/OLAP systems, while HTAP benchmark can not be a simple stitch of OLTP/OLAP benchmark [21]. For OLTP benchmarks like TATP [47], TPC-C [48], and TPC-E [51], we often fix transaction templates and execution logic to measure execution efficiency per unit of time under a specified access distribution. For OLAP benchmarks like SSB [34], TPC-H [50], and TPC-DS [49], we generally fix query predicate parameters under a relatively static data set. Because of the richer combination of operators and the larger number of tables involved, we are more concerned about the validity of data cardinality under deep joins and complex predicate conditions. Similarly, we evaluate the OLAP performance by the number of queries executed per unit of time, i.e., QphH.

Nevertheless, besides OLTP and OLAP services provided by HTAP systems, the main distinct characteristic of HTAP systems is the consumption requirement of fresh data, which means the overlapping of data access between OLTP and OLAP workloads [21]. It implies that 1) the **computational complexity** of an OLAP workload is highly affected by the produced data from T-engine. Additionally, HTAP systems may store two different formats of data to serve OLTP/OLAP services separately [15, 17, 20, 24]. It implies that 2) the data interleaving degree of OLAP workloads with the latest modification from OLTP workloads makes **synchronization and data reorganization cost** dynamically change. Though in existing benchmarks, the HTAP business features have been considered to some extent, like introducing real-time queries within transactions, the more critical issues caused by runtime OLTP modifications are the dynamic changes of OLAP computational complexity, data synchronization and reorganization costs arising from the different scales of fresh data access, which have not been seriously considered. To facilitate a fair comparison, it is essential to quantify or control these dynamics between HTAP workloads.

Challenges in HTAP Benchmark Design. In practice, designing a fair HTAP benchmark is challenged by the following three main conflicts:

- **Distinct Processing Capacity** *vs*. **Consistent Workload Complexity.** In an HTAP system, T-engine produces data and A-engine consumes data. However, T-engine of HTAP systems can expose distinct OLTP processing capacities, e.g., TPS, and then generate different volumes of data at the same time interval. As shown in Fig. 1, we monitor data increasing (row count) of frequently written tables in TPC-C, i.e., *Order* and *Orderline*, 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 to different A-engines will meet totally different computational complexity at the same time point. Moreover, distinct capabilities of A-engines can lead to differences in execution latencies for the same query. Then even if the T-engines present the same TPS, they will access different sizes of data (synchronized) from the second query on different A-engines. 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 pick one of the characteristic HTAP system PostgreSQL-SR as an example and conduct experiments by selecting two complex representative queries respectively in the widely adopted CH-benchmark [13] and in the state-of-the-art HATtrick [32]. We configure 8 OLTP threads and 8 OLAP threads, run for 900 seconds and uniformly randomly pick 40 query latency results to expose the latency stability. It is obvious that the query latency is highly unstable, causing low benchmarking repeatability. The reason mainly arises from the lack of an associated control in query parameter population and data increase. Thus, the distinct capacity of T- and A-engines makes it tough to align the data to have the same computational complexity.

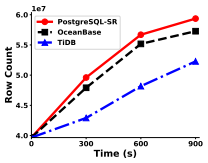
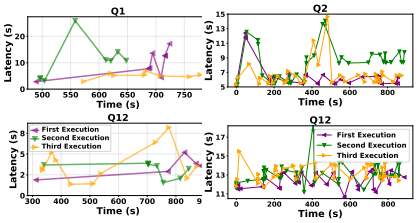


Figure 1: Data Increase in Three Systems



(a) CH-benchmark (b) HATtrick

Figure 2: Stability of Query Latency

- **Nondeterministic Data Access Interleaving** *vs*. **Deterministic Interaction Pattern Orchestrating.** HTAP systems require the newly generated data of T-engine to be accessed by A-engine within a freshness threshold. It is easy to see that the degree of freshness and the volume of freshness data accessed by A-engine have an obvious impact on the performance of HTAP systems, which produce variant synchronization pressures. Therefore, quantifying the synchronization cost or orchestrating this nondeterministic overlap/interleaving relationship of mixed workloads into a deterministic and controllable interaction pattern for various query scenarios is the core challenge in evaluating the real-time processing capability.
- **Multi-dimensional Metrics** *vs*. **Unified Comparability.** Different applications have various freshness requirements, while

high freshness is at the cost of performance [45], which inspires the existence of various architectures to provide diverse data freshness for analysis and achieve a performance-freshness trade-off. So the observed 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 to launch a comprehensive capability comparison among HTAP systems.

To deal with the challenges of benchmarking HTAP systems, we propose a new benchmarking framework called *Vodka* with its contributions summarized as follows. (1) We are the first work to formalize query parameter regulation to various operators with the purpose of query computation complexity controlling under dynamic data changes caused by *T*-engine (§3). We then provide a linear model to query latency and the data size. (2) *Data Sharing Model* defines data consistency degree between *T*- and *A*-engines, which usually trades off the performance and the freshness. We propose a novel query-oriented freshness definition by a unified metric to measure the ability of HTAP systems for user current requests instead of the global data freshness (§??). (3) We conduct extensive experiments on three HTAP systems, i.e., PostgreSQL Streaming Replication [41], OceanBase [60], and TiDB [20], corresponding to different data sharing model on different data storage architectures (§6). We summarize some interesting observations, which are….

2 BACKGROUND

In this section, we first study the distinct key techniques of an HTAP system for *resource isolation* and *data sharing*. Then, we analyze the benchmarking ability of the existing popular HTAP Benchmarks *w.r.t* these technical designs, which motivates to present *Vodka*.

An HTAP system usually includes two independent engines [32], i.e., OLTP engine (*T*-engine) and OLAP engine (*A*-engine), each of which can have multiple physical instances. *T*-engine generally executes transactions based on the small size of data for the high throughput, i.e., TPS. In contrast, *A*-engine typically involves large-scale data scans, complex joins, or aggregates, which are computation intensive operations and consume much CPU, memory, disk IO/storage, or network IO and have a relatively high query latency. Therefore, executing hybrid workloads in a single system will naturally cause resource contention which leads to performance degradation for both *T*- and *A*-engines. 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**. Meanwhile, *T*-engine keeps producing data which is expected to be consumed by *A*-engine in a limited time period (a.k.a the degree of data *freshness*) [32]. If *A*-engine can read the latest production data from *T*-engine, it provides the most valuable data, i.e., the highest *freshness*. However, to provide the data freshness in HTAP systems usually imposes a severe data organization or communication demand to *T*-engine [31]. Therefore, to guarantee the performance of each engine, **the other task for an HTAP system is to provide an efficient *data sharing* model** to ensure data freshness [53].

To conclude, compared to the original single-type systems, HTAP systems devote the most effort to **achieving efficient data sharing under effective resource isolation**.

Table 1: Resource Isolation Techniques

Resource Isolation				
Isolation Degree		**Logical**	**Memory**	**Storage**
Type	**CPU**	Virtualization [30];	Physical Isolation	Physical Isolation
	Memory	Delta-Versioning [15] MVCC [23]; CoW [22]		
	Disk	/	/	
Representative Systems		Oracle [24]; Hyper [22] OceanBase [59]	PolarDB [7] Aurora [52]	TiDB [20] BatchDB [31]

2.1 Resource Isolation

Resource Isolation refers to scheduling and separating the hardware resources, i.e., CPU, memory, disk IO/storage, etc., for *T*- and *A*-engine. We divide the classic implementation techniques into three categories, i.e., *logical isolation*, *memory isolation*, and *storage isolation* as in Table. 1.

For Logical Isolation, both *T*- and *A*-engines share all resources of the same machine, and we use logical methods to schedule resources [44] as shown in Fig. 3 ❶. For CPU isolation, it uses virtualization techniques to split CPU resources into groups *w.r.t* hybrid workloads, e.g., in Greenplum [30]. For memory isolation, according to the granularity of data versions, the techniques can be further classified into three types. The first one is to maintain two data replicas, i.e., *Main* and *Delta* versions, for serving OLAP reads and OLTP writes respectively as in SAP HANA [15], Hyrise [17] and Oracle [24]. Meanwhile, to catch up with the high throughput of *T*-engine, *A*-engine has to fetch updates from heterogeneous OLTP replica (*Delta* version), causing extra data reorganization costs. The second one is based on the *copy-on-write(COW)* mechanism [4] as in Hyper[22], which reads by creating the latest snapshot of data independently without blocking OLTP updates. However, obtaining fresh data requires frequent snapshot creation, which may occupy huge memory resources especially for massive parallel OLAP reads. The last option is to share the *multiversion* of tuples [57] created by *T*-engine to *A*-engine as in PostgreSQL [**?**]. As both engines share a unified version chain, it avoids the expensive cost of data copy or synchronization. However, *A*-engine may meet an expensive traverse cost to identify the consistent versions of tuples for a query among the involved version chains [23, 45].

For Memory Isolation, both *T*- and *A*-engines share storage resources in the same machine, but CPU and memory resources are isolated by placing OLTP and OLAP tasks to different machines [52] as shown in Fig. 3 ❷, which maintains memory snapshots independently as in PolarDB [7] and Aurora [52]. But OLAP clients may maintain different snapshot versions of data, which inevitably leads to the problem of data inconsistency between the memory and storage layers.

For Storage Isolation, *T*- and *A*-engines are put separately on different machines, which has an absolute hardware isolation as shown in Fig. 3 ❸. *T*-engine synchronizes its logs or data to *A*-engine for fresh data accessing as in TiDB [20], BatchDB [31] and

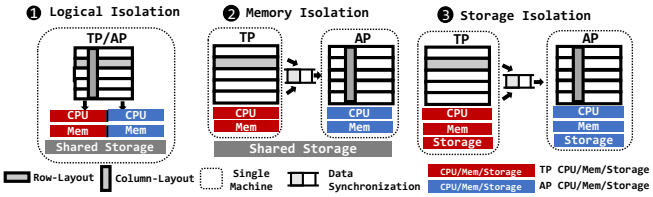


Figure 3: Overview of Resource Isolation

F1 Lightning[58], but it does not guarantee the real-time data consistency between the two types of engines. For the fresh data, *A*-engine will need to ask *T*-engine to transfer the latest updates, which meets a high network cost and magnifies the query latency.

2.2 Data Sharing

Data sharing refers to the method of sharing the produced data from *T*-engine to *A*-engine [46]. It covers the whole lifecycle of data, including allocating data version timestamp, synchronizing/organizing data, and tracing data for reading. Note that the latest OLTP write versions and the OLAP read versions may not be equal. That is there exist version gaps or time delays between *T*-engine and *A*-engine for the same data. The gap follows the definitions of **data sharing models** (*DSM*s, details in §4.1), which provides various data consistency to guarantee different freshness degrees, including *linear consistency*, *sequential consistency, and session consistency*. Suppose we have one OLTP instance and one or more OLAP instances. The time stamp oracle (TSO) [37] is responsible for allocating data version timestamp. *T*-engine's TSO can be synchronized to *A*-engine, based on which OLAP query can read the expected version snapshot.

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 [15], Hyrise [17], TiDB [20], Oracle [24], SQL Server [27], Greenplum [30], SingleStore [42] and OceanBase [59]. Both *T*- and *A*-engine share the same TSO and read the consistent latest data version. In practice, it's usually implemented on a unified storage database [15, 17, 24, 59]. 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 [20].

Sequential Consistency makes a concession on data freshness for *A*-engine to avoid OLTP performance degradation for synchronizing data to *A*-engine. It can access previous snapshot versions (stale data), but all *A*-engines should read the consistent same version of the snapshots at the same time-point. It is adopted by IDAA [6], BatchDB [31] and Vegito[45]. All OLAP instances share the same TSO which is behind OLTP's TSO. In practice, most of them are used for HTAP systems with decoupled storage, which synchronize OLTP data in epochs (batches).

Session Consistency discards the constraint to see the consistent same version on all OLAP instances but only requires each client to see data following the strict version generation order from *T*-engine, supported by PolarDB [7], Aurora[52] and Hyper[22]. Every OLAP instance can have its independent TSO, which is behind OLTP's TSO, and is empowered to have different data versions. Thereby it favors high performance at the cost of data freshness. 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.

2.3 Remarks for Benchmarking HTAP Systems

Compared to traditional single-type database systems, HTAP systems have their distinct characteristics of providing both OLAP and OLTP services 1) under **a unified business model**; 2) **resource isolation** for stable service; 3) **data sharing** for fresh data consumption, and the tailored 4) **metrics** for these new characteristics. According to these requirements, we identify the limitations in Table 2 of current mainstream HTAP benchmarks, including CH-benCHmark [13], HTAPBench [12], Swarm64 [40], OLxPBench [21], and HATtrick [32].

HTAP systems are deployed to sever applications with hybrid workloads. Under the *unified application scenario*, it is imperative to guarantee semantic consistency for both *T*- and *A*-engines, including table schema, data distribution as well as workload execution logic on the same business data. Most of the benchmarks use the stitched schema from existing benchmarks. For example, CH-benCHmark, HTAPBench, and Swarm64 are based on TPC-C and TPC-H considering little semantics consistency. The deficiency is first mentioned by OLxPBench [21] in the year 2022. 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/-complex predicates make the performance evaluation incomprehensive. Likewise, HATtrick constructs simple OLTP transactions based on SSB (a.k.a. simplified TPC-H). The simplicity of both OLTP and OLAP workloads makes it not a representative HTAP application scenario.

HTAP systems use *resource isolation* technique to mitigate resource contention between OLTP and OLAP clients, with the purpose of reducing performance interference and guaranteeing stable OLTP/OLAP performance [44]. Then we define the problem of benchmarking resource isolation to achieve a fair measurement of OLTP/ OLAP performance and a precise measurement of mutual interference degree. For the former, as far as we know, no-existing work takes different OLTP throughputs and uncertain synchronization progress among systems into consideration, which will cause OLAP queries to have naturally unequal data size and uncertain inter-column joint probability distribution. Then query computation complexity varies uncontrollable or uncomparable among different HTAP systems. So the benchmark results can have obvious bias from different rounds of runs for different HTAP systems even for the same one. To compare the interference between T- and A-engines, it usually uses the observation of OLTP/OLAP performance degradation degree when adding more OLAP/OLTP threads. It is adopted by HATtrick and OLxPBench and is imprecise. The reason is that the degradation of OLTP performance means the various decreases of data size expansion speed causing the bias of resource consumption for OLAP queries.

No matter how isolation is, *T*-engine should share data with *A*-engine for various freshness requirements by defining different *data sharing models* [31]. Meanwhile, users focus more on whether the data in the access range of their queries is fresh enough [5, 38],

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**
Uniform Application Scenario		Semantically-Consistent Benchmark Scenario	F	F	F	T	T	T
		Representative OLAP&OLTP Workloads	T	T	T	P	P	T
Key Techniques	**Resource Isolation**	Fair TP/AP 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

instead of the global freshness state of the whole system. Therefore, the benchmark has the urgent demand to have a query/user-oriented freshness calculation method and to regulate data sharing pressure. To the best of our knowledge, current benchmarks all fail to adequately cover these requirements. Though CH-benCHmark has provided a definition of freshness, it lacks an exact measurement method. HATtrick's freshness score focuses on the global freshness state of the system. Meanwhile, none of them proposes to quantify the data sharing pressure derived from the interaction between mixed workloads.

A mainstream categorization of HTAP system architectures is based on the sharing of storage, namely *unified* and *decoupled* storage [44]. As far as we know, there has been no work that proposes a system classification based on *data sharing models*, i.e., data consistency between *T*- and *A*-engine. This lack of architectural perspective results in existing benchmarks assuming that data in the *T*- and *A*-engines is consistent, leading 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.

After a thorough study of the benchmark ability *w.r.t* the core design technology in HTAP systems, we observe that none of the existing works 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 fair comparison among HTAP systems.

3 HYBRID WORKLOAD CARDINALITY 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 [24, 31, 45]. We follow the same way, and then all HTAP systems are assumed to have the same increase of new data after the same time. With this precondition, we present our approach to quantify the cardinalities of SPJ operators, i.e., selection, join, projection, for a given data size in § 3.1. It then facilitates the modeling of query latency under different data sizes in § 3.2 to achieve a fair comparison among HTAP systems with different workload processing ability.

3.1 Query Cardinality Control

Repeatability has been a fundamental requirement in traditional OLAP benchmarks, that is different rounds of benchmarking expose almost the same performance. It is guaranteed by keeping a stable computation complexity of operators, which is closely related to the operator cardinality [29]. However, in HTAP scenarios, the dynamic data changes derived from OLTP modifications with different consistency requirements introduce uncertainty in the output cardinality of operators, resulting in unpredictable query complexities or unstable query latency during the benchmarking period.

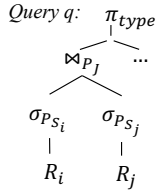


Figure 4: An Illustration of Query Logical Tree

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 as in Fig. 4. 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 operator for aggregations is denoted as π_{type}. Since the projection is often at the top node of a query tree, we prioritize its cardinality control after that of join and selection operators.

Note that all the state-of-the-art and state-of-the-practice OLAP benchmarks [49, 50] 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. 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 Equation 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 Equation 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 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* [**?**], the **SEL**ectivity (*Sel*) of the output from the join ($\bowtie$) and selection (σ) operators can be generalized by Equation 3, which represents the probability (*Pr*) of generating results from R_i and R_j satisfying selection and join predicates, i.e., P_S and P_J. Meanwhile, Equation 3 can be viewed as the probability of the situation where both the selection and join predicates are satisfied regardless of the concrete execution order of them so that it is then simplified to Equation 4. According to Bayes' theorem, Equation 4 can be equally represented by Equation 5. Based on the σ-Indep and $\bowtie$-Indep, we can further deduce Equation 5 to Equation 6 and Equation 7, respectively.

$$Sel(\bowtie,\sigma) = Pr(\sigma_{P_{S_i}} \bowtie_{P_J} \sigma_{P_{S_j}}) \tag{3}$$
$$= Pr(\sigma_{P_{S_i}}, \sigma_{P_{S_j}}, \bowtie_{P_J}) \tag{4}$$
$$= Pr(\sigma_{P_{S_i}}, \sigma_{P_{S_j}} \mid \bowtie_{P_J}) * Pr(\bowtie_{P_J}) \tag{5}$$
$$= [Pr(\sigma_{P_{S_i}} \mid \bowtie_{P_J}) * Pr(\sigma_{P_{S_j}} \mid \bowtie_{P_J})] * Pr(\bowtie_{P_J}) \quad //\sigma_{P_{S_i}} \perp \sigma_{P_{S_j}} \tag{6}$$
$$= Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * Pr(\bowtie_{P_J}) \quad //\sigma_{P_{S_i}} \perp \bowtie_{P_J}, \sigma_{P_{S_j}} \perp \bowtie_{P_J} \tag{7}$$

Then the **CARD**inality (*Card*) from the join and selection is calculated by Equation 8.

$$\begin{aligned} Card(\bowtie,\sigma) &= Sel(\bowtie,\sigma) * |R_i| * |R_j| = Pr(\sigma_{P_{S_i}} \bowtie_{P_J} \sigma_{P_{S_j}}) * |R_i| * |R_j| \\ &= Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * (Pr(\bowtie_{P_J}) * |R_i| * |R_j|) \\ &= Pr(\sigma_{P_{S_i}}) * Pr(\sigma_{P_{S_j}}) * |R_i \bowtie_{P_J} R_j| \end{aligned} \tag{8}$$

It means that 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, i.e., $Pr(\sigma_{P_{S_i}})$ and $Pr(\sigma_{P_{S_j}})$, from selection predicates. Even though we can ensure the aforementioned mutual independencies, data is evolving due to the *T*-engine in an HTAP system, which increments $\Delta_{i/j}$ for $R_{i/j}$ and becomes $R'_{i/j}$. 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 as Equation 9.

$$\begin{aligned} Card(\bowtie,\sigma) &= \mathbf{Pr}(\sigma_{P_{S_i}}) * \mathbf{Pr}(\sigma_{P_{S_j}}) * |\mathbf{R'_i} \bowtie_{P_J} \mathbf{R'_j}| \\ s.t. \quad R'_i &= R_i \cup \Delta R_i \text{ and } R'_j = R_j \cup \Delta R_j \end{aligned} \tag{9}$$

Finally, the output cardinality in Equation 9 is related to the selectivity of selection predicates $Pr(\sigma_{P_{S_{i/j}}})$ and the join cardinality of join predicate P_J on relations R'_i and 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 operator (in §3.1.3).

3.1.1 Selection Selectivity Control. Let us first analyze the selectivity of selection predicates, e.g., $Pr(\sigma_{P_{S_i}})$ and $Pr(\sigma_{P_{S_j}})$, and find ways to make it predictable even in the dynamic scenarios. Suppose the selectivity α ($0 \le \alpha \le 1$) is calculated based on initial query parameters. Take a selection predicate '$l_i \le A_i \le r_i$' 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) generates various α.

The naive method is to scan the database and fill the parameters for a target α. Besides the high traversal cost in a large-scale relation, the high throughput of *T*-engine makes the data dynamically change, and then the scan-based parameter instantiation is impossible for our target. Reservoir sampling [1, 3, 55] 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, where $N = \frac{ln2 - ln(1-\hat{\rho})}{2\delta^2}$ [18] with $\delta, \hat{\rho} \in [0, 1]$.

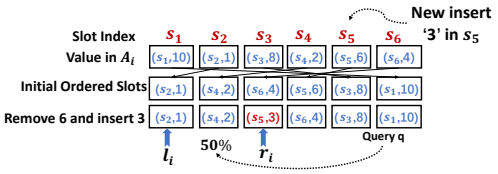


Figure 5: An Example of Reservoir Sampling

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 Equation 10.

$$P_{S_i} = \wedge_{x=1}^{n} clause_x \quad s.t. \quad clause_x = \vee_{y=1}^{m} literal_{xy}. \tag{10}$$

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 $literal_{xy}$ 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 $\bullet \in \{=, \neq, <, >, \le, \ge, (not)\ in, (not)\ like\}$, 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 $g(A_i, ..., A_k) \circ val$, $\circ \in \{<, >, \le, \ge\}$, 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 cases.

- The simplest predicate is P_{S_i}=$literal$, i.e., n=1 and m=1 in Equation 10. The predicate is generalized as P_{S_i}=$l_k \le A_k \le r_k$. 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 randomly 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}.
- A predicate has n $clause$s, each of which has only one $literal$, i.e., $P_{S_i} = \wedge_{k=1}^{n} l_k \le A_k \le r_k$, that is $n > 1$ and m=1 in Equation 10. If $A_{k_1}, \cdots, A_{k_v}$ are independent, we maintain the reservoir for each column A_{k_j} ($k_1 \le k_j \le k_v \le n$), 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 (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. Even when literal has an arithmetic computation as P_{S_i}=$g(A_{k_1}, \cdots, A_{k_v}) \le r$, it constructs the reservoir based on the results of $g()$ function on the v columns, with the entry as $(si, g(A_{k_1}, \cdots, A_{k_v}))$, which are used to instantiate the parameters related to $g()$.
- A predicate has one $clause$ with more than one $literal$s, i.e., P_{S_i} =$\vee_{k=1}^{m} l_k \le A_k \le r_k$, that is n=1 and $m > 1$ in Equation 10. It can be transformed to the second case, i.e., P_{R_i} =$\wedge_{k=1}^{m} l_k \le A_k \le r_k$, but with the selectivity $(1-\alpha)$.
- A predicate has several $clause$s with more than one $literal$s, i.e., $n > 1$ and $m > 1$ in Equation 10. Since the core target is to guarantee the selectivity α, we draw inspiration from *Set Theory* (i.e., $Rule_1$ and $Rule_2$) to lower down the complexity by reducing instantiating parameters [28, 33] with domain boundary values.

$$Rule_1 : literal_i \cup literal_j = literal_j \quad if \quad literal_i \leftarrow \emptyset$$
$$Rule_2 : clause_i \cap clause_j = clause_j \quad if \quad clause_i \leftarrow U$$

Suppose the example predicate component of the $literal/clause$ is formatted as $l_k \le A_k \le r_k$. $Rule_1$ 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 $m > 1$ (Detailed proof is listed in our technical report [?]). 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 $clause_2$.

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

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

To conclude, reservoir sampling can help to sample the dynamic data and locate the parameters 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.

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 shown in Fig. 6. We take the primary-foreign key (PK-FK) equal join between the referenced relation R_i and referencing relation R_j to make a discussion with the data increasing as ΔR_i and ΔR_j. This method can be generalized to any other join type.

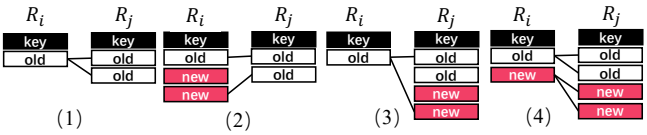


Figure 6: Illustration of Increment Joins

Case 1 ΔR_i=$\emptyset$ & ΔR_j=$\emptyset$**:** Join involves no new data as in Fig. 6 (1). The join cardinality size always keeps the same, i.e., $|R_i \bowtie_{P_J} R_j| = |R'_i \bowtie_{P_J} R'_j|$.

Case 2 $\Delta R_i \neq \emptyset$ & $\Delta R_j = \emptyset$**:** Only the referenced relation R_i increases by $\Delta R_i (\neq \emptyset)$ as in Fig. 6 (2). 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., $|(R_i \bowtie_{P_J} R_j) \cup (\Delta R_i \bowtie_{P_J} R_j)| = |(R_i \bowtie_{P_J} R_j) \cup \emptyset| = |R_i \bowtie_{P_J} R_j|$

Case 3 $\Delta R_i = \emptyset$ & $\Delta R_j \neq \emptyset$**:** Only the referencing relation R_j increases by $\Delta R_j (\neq \emptyset)$ as in Fig. 6 (3). Since we ensure that newly inserted keys follow the same distribution as the initial keys, it ensures that the increment join cardinality from ΔR_j is proportional to R_j, i.e., $|(R_i \bowtie_{P_J} \Delta R_j)| = \frac{|\Delta R_j|}{|R_j|} |(R_i \bowtie_{P_J} R_j)|$. Therefore, we have $|R_i \bowtie_{P_J} (R_j \cup \Delta R_j)| = |(R_i \bowtie_{P_J} R_j) \cup (R_i \bowtie_{P_J} \Delta R_j)| = |(R_i \bowtie_{P_J} R_j)| + |(R_i \bowtie_{P_J} \Delta R_j)| = |(R_i \bowtie_{P_J} R_j)| + \frac{|\Delta R_j|}{|R_j|} |(R_i \bowtie_{P_J} R_j)| = (1 + \frac{|\Delta R_j|}{|R_j|}) |(R_i \bowtie_{P_J} R_j)|$.

In Case 3, the join cardinality change is linearly related to ΔR_j, which can be quantified by the throughput of T-engine.

Case 4 $\Delta R_i \neq \emptyset$ & $\Delta R_j \neq \emptyset$**:** Both the referenced and referencing relations increase as in Fig. 6 (4). The join cardinality can be first deduced based on *Distributive Law* as in Equation 11.

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

The cardinality increment is from $|[R_i \cup \Delta R_i] \bowtie_{P_J} \Delta R_j|$, where the number of foreign keys in ΔR_j can come from either R_i or ΔR_i, represented by $|\Delta R_j^o|$ and $|\Delta R_j^n|$ respectively, and $|\Delta R_j| = |\Delta R_j^o| + |\Delta R_j^n|$. Therefore, Equation 11 can be further transformed into Equation 12.

$$\begin{aligned} &|R_i \bowtie_{P_J} R_j| + |(R_i \cup \Delta R_i) \bowtie_{P_J} \Delta R_j| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \cup \Delta R_i) \bowtie_{P_J} (\Delta R_j^o \cup \Delta R_j^n)| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \bowtie_{P_J} \Delta R_j^o) \cup \emptyset \cup (\Delta R_i \bowtie_{P_J} \Delta R_j^n) \cup \emptyset| \\ &= |R_i \bowtie_{P_J} R_j| + |(R_i \bowtie_{P_J} \Delta R_j^o) \cup (\Delta R_i \bowtie_{P_J} \Delta R_j^n)| \end{aligned} \tag{12}$$

For ΔR_j^o is only related to R_i and it is semantics consistent with R_j in data distribution, $|(R_i \bowtie_{P_J} \Delta R_j^o)|$ has the same result as Case 3, i.e., $|(R_i \bowtie_{P_J} \Delta R_j^o)| = |\frac{|\Delta R_j^o|}{|R_j|} (R_i \bowtie_{P_J} R_j)|$, which is linearly related

to the original join cardinality. To obtain a stable increasing of $|\Delta R_i \bowtie_{P_J} \Delta R_j^n|$, an *equal-expansion-ratio-restricted joint distribution* between R'_i and R'_j can help to guarantee our expectation as defined in Equation 13, which is popularly adopted by current OLAP benchmarks [50].

$$\frac{|\Delta R_i|}{|R_i|} = \frac{|\Delta R_j^n|}{|R_j|} = \frac{|\Delta R_i \bowtie_{P_J} \Delta R_j^n|}{|R_i \bowtie_{P_J} R_j|} \tag{13}$$

Combine Equations 12 and 13 and then we have Equation 14.

$$\begin{aligned} &\frac{|\Delta R_j^o|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| + \frac{|\Delta R_j^n|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| \\ &= \frac{|\Delta R_j^o| + |\Delta R_j^n|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| = \frac{|\Delta R_j|}{|R_j|}|(R_i \bowtie_{P_J} R_j)| \end{aligned} \tag{14}$$

Finally, for Case 4, it has a linear increase as Equation 15.

$$|(R_i \cup \Delta R_i) \bowtie_{P_J} (R_j \cup \Delta R_j)| = (1+\frac{|\Delta R_j|}{|R_j|})|R_i \bowtie_{P_J} R_j| \tag{15}$$

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 Cardinality Control. According to the computational complexity, *projection* π can be grouped into two classes. The complexity of the first class is related to the input cardinality size (i.e., $card_s$), such as COUNT and SUM, while the second one is related to the distinct cardinality of input (i.e., $card_d$), such as GROUP BY. Since π usually happens after *join* and *selection*, the input cardinality size $card_s$ of π 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 π, which may greatly influence the computational complexity of the second case, e.g., GROUP BY.

Notice that projection on non-key column π_{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. Such categorized data has a limited impact on performance [28]. Projection on key columns happens only on foreign keys (FKs) denoted as π_{fk}, for primary keys (PKs) are distinct and used as the identifiers of rows. Therefore, $Card_d$ of the FK column is from the dependent PK column of the referenced relation, which is unpredictable. Fortunately, suppose we launch a π_{fk} operation after $\bowtie_{P_J}$ as in Equation 16, the result is equal to the set of joinable PKs of the referenced table, which can be achieved by a left semi-join ($\ltimes_{P_J}$) [54], and finally it is represented by Equation 17.

$$\pi_{fk}((R_i \cup \Delta R_i) \bowtie_{P_J} (R_j \cup \Delta R_j)) = (R_i \cup \Delta R_i) \ltimes_{P_J} (R_j \cup \Delta R_j) \tag{16}$$

$$\begin{aligned} &= |R_i \ltimes_{P_J} R_j| + |(R_i \ltimes_{P_J} \Delta R_j^o) \cup (\Delta R_i \ltimes_{P_J} \Delta R_j^n)| \\ &= |R_i \ltimes_{P_J} R_j \cup (R_i \ltimes_{P_J} \Delta R_j^o)| + |(\Delta R_i \ltimes_{P_J} \Delta R_j^n)| \\ &= |R_i \ltimes_{P_J} R_j| + |\Delta R_i \ltimes_{P_J} \Delta R_j^n| \end{aligned} \tag{17}$$

$$= |R_i \ltimes_{P_J} R_j| + \frac{|\Delta R_i|}{|R_i|}|R_i \ltimes_{P_J} R_j| = (1+\frac{|\Delta R_i|}{|R_i|})|R_i \ltimes_{P_J} R_j| \tag{18}$$

Since we have followed the *equal-expansion-ratio-restricted joint distribution* requirement to the increasing of join tables, we have $(\Delta R_i \ltimes_{P_J} \Delta R_j^n) = \frac{|\Delta R_i|}{|R_i|}|R_i \ltimes_{P_J} R_j|$. A linear expansion of $Card_d$ on FKs can still be achieved, as represented by Equation 18.

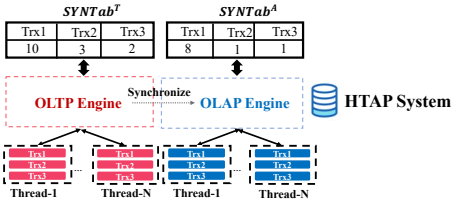


Figure 7: Data Size and Query Latency Modeling

3.1.4 Cardinality Control for SPJ. To conclude, we can achieve a **linear expanding of the cardinality from the SPJ operators** (**S**election-**P**rojection-**J**oin) along with the data increasing over time, which is produced by T-engine, so that query complexity is controllable, which facilitates the benchmarking repeatability for HTAP systems. Moreover, a linear cardinality expansion enables systems to maintain a relatively stable query plan.

3.2 Data Size-based Latency Modeling.

We have achieved a linear increase of each operator's cardinality in a query as data grows. To launch a fair benchmarking among HTAP systems, it is imperative to compare the performance under the same input data size. However, even for the same throughput from the T-engines of HTAP systems, there still exist two obstacles 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 providing various data consistency can see **different sizes of data** at the same time point.

Note that the execution cost of a query is closely related to the cardinality of each operator [29] and a linear cardinality change can have a linear query latency change when the query plan remains the same. Since we have controlled to have a stably linear cardinality increasing of SPJ operators, it then motivates us to build a linear regression model $L(S) = k \cdot S + b$ to predicate query latency, where $L(S)$ is the query latency under the data size S, k and b are two fitting parameters.

To obtain the real-time data size S for A-engine and avoid costly scan in relations, we propose to create transaction synchronization tables (*SYNTab*) on both sides of the T- and A-engine as shown in Fig. 7, which are $SYNTab^T$ and $SYNTab^A$ respectively. In $SYNTab^T$, it records the number of transactions (Trx) completed and in $SYNTab^A$, it presents the modifications (from Trx) that have been synchronized. If it does not require a strict data synchronization (consistency) between T- and A-engine in HTAP systems, transaction synchronized to $SYNTab^A$ usually lags behind the ones executed in $SYNTab^T$. For example, in Fig. 7, 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. For example, in TPC-C, a committed *NewOrder* transaction will insert 10 rows into relation *Orderline* on average, so the number of committed *NewOrder* transactions $\#Trx_{NewOrder}$ synchronized can deduce the data increment of Orderline by $10 \cdot \#Trx_{NewOrder}$.

In practice, $SYNTab$ is a lightweight $TransactionInfo$ 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 [32], 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 performance fluctuations from network or other factors, we launch the regression fitting approach based on *Huber Loss* function (defined in Equation 19), which is insensitive to noise and serves as an effective method for deciding fitting parameter of k and b in $L(S)$ [8, 19, 26].

$$Huber\ Loss(\theta,\mu) = \begin{cases} \frac{1}{2}\mu^2, & \text{if } |\mu| \le \theta, \\ \theta(|\mu| - \frac{1}{2}\theta), & \text{otherwise.} \end{cases} \tag{19}$$

Specifically, $\mu = L(S) - \hat{L}(S)$ represents the residual between real latency $L(S)$ and predict latency $\hat{L}(S)$ (=$k \cdot S + b$) 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} can be used to fit the best k_i and b_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 $QphH$ [49, 50] to demonstrate the overall OLAP performance but redefine it by Equation 20. 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 same data size S.

$$QphH = 3600 * \frac{W * TH}{\sum_i^W L_i(S)} \tag{20}$$

4 QUERY-ORIENTED DATA FRESHNESS

In this section, we first formalize data sharing models providing variant data consistency in §4.1; then we give the definition of query-oriented data freshness in §4.2, which focuses on calculating freshness for each query instead of the global database system, and design a unified metric by integrating both the freshness and query latency; finally, we propose to quantify data sharing pressure by introducing workload interaction pattern to control sharing data size in §4.3.

4.1 Data Sharing Models

As we have mentioned in § 2.2, 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. 8. 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 Trxs, i.e., $w_1 \cdots w_4$, and the other clients launch read Trxs. For simplicity, the following introduction of data version evolution is from the most recent write of T-engine, i.e., checkpoints, labeled as t_i. Before we define *DSM*, we introduce some concepts for version evolutions.

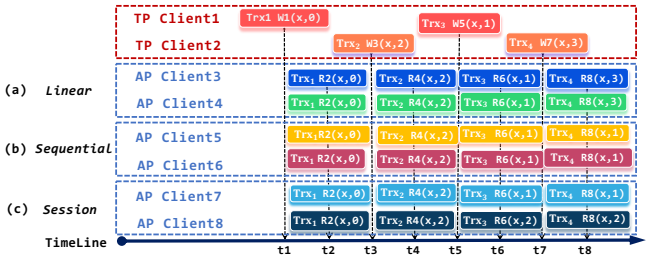


Figure 8: Read/Write x under Different Data Sharing Models

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. 8, $Client_1$ 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. 8, the version write sequence S_{t_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 $t_p \le t$, 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 $S_{t_8} = \{w_1, w_2, w_3, w_4\}$ at t_8, its prefix write sequence can be $S_{t_8}^{p_1} = \{w_1\}$, $S_{t_8}^{p_2} = \{w_1, w_2\}$, $S_{t_8}^{p_3} = \{w_1, w_2, w_3\}$, $S_{t_8}^{p_4} = \{w_1, w_2, w_3, w_4\}$.

Data Version: A data version v_j of x is the result of linearly executing a prefix write sequence $S_{t_j}^p$. The version-stamp of x at t_j from t_0 is defined by the length of its prefix write sequence, i.e., $|S_{t_j}^p|$. $Len[v_{t_j}(x)]$ denotes the version-stamp of x at t_j from t_0 defined by the sequence length, i.e., $Len[v_{t_j}(x)]$=$|v_{t_j}(x)|$.

Example 4. For $Client_3$ at t_6, the result of executing $S_{t_6} = \{w_1, w_2, w_3\}$ on x serially is 1, which is a sub-sequence of S_{t_8}. The data version and data version-stamp is $v_{t_6}(x)$=1 and $|S_{t_6}|$=3, respectively.

Version Read: For a read $r_i(x, v)$ of a session (simplified as r_i=v) at time $r_i.t$, it has the monotonicity of its read sequence, i.e., $\forall r_i.t \le r_j.t, |S_{r_i.t}| \le |S_{r_j.t}|$.

Example 5. For $Client_3$, Txn_3 reads the data version of x at t_6, i.e., r_{t_6}=1, from a write sequence S_{t_5}, while r_{t_4}=2 is from its prefix write sequence S_{t_3}, among which $S_{t_3} \subseteq S_{t_5}$.

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 gap of visible versions between *T*- or *A*-engines. Specifically, DSM_{TA} (resp. DSM_{AA}) in Equation. 21 (resp. Equation. 22) declares the gap between *T*- and *A*-engines (resp. between reads $r_{i/j}$ from different sessions $s_{i/j}$ of *A*-engines). Based on the differences of the data version gaps, represented by p and q in Equation. 21 and 22, we summarize three *DSMs* as followings.

$$DSM_{TA}(v_{w.t}, v_{r.t}) = |L[v_{w.t}(x)] - L[v_{r.t}(x)]| = p,\ s.t.,\ w.t = r.t \quad (21)$$

$$DSM_{AA}(v^{s_i}_{r_i.t}, v^{s_j}_{r_j.t}) = |L[v^{s_i}_{r_i.t}(x)] - L[v^{s_j}_{r_j.t}(x)]| = q, s.t., r_i.t = r_j.t \quad (22)$$

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 $S^p_{r.t}$, 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^s_r is produced by executing a prefix write sequence of $S^s_{r.t}$ 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.

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 [20] OceanBase [60] Oracle [24]	F1 Lightning [58] BatchDB [31] Vegito [45]	PolarDB [7] Aurora [52] Hyper [22]

4.2 Freshness Calculation

Intuitively, data freshness refers to how up-to-date the accessed data is [5], and in previous work, measuring data freshness typically includes three main approaches as shown in Fig. 9.

First, it can be calculated by the overlap ratio of identical tuples between the *T*- and *A*-engine [44]. In Fig. 9 ❶, it has overall three tuples in both *T*- and *A*-engine and there exists one tuple that is different on these two types of engines (painted in red). So the freshness by the overlap ratio of identical tuples is 2/3. However, it's expensive to scan and compare all data tuples in a large-scale database system. Second, in Vegito [45] and ByteHTAP [9] systems, freshness is calculated based on the time elapsed from the latest data version generated by the *T*-engine to its synchronization to the *A*-engine [5, 11], as shown in Fig. 9 ❷. However, the detailed synchronization time duration is generally obtained in the system internally and hardly acquired externally when benchmarking. Finally, HATtrick [32] 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 to *A*-engine at its query start time t^s_q, i.e. $freshness = max(0, t^s_q - t^{fns})$. Specifically, in Fig. 9 ❸, to check the first not-seen transaction, it maintains a freshness table to record the latest transaction ID (monotonically increase) for each T-client, e.g., $F_1, \ldots, F_n$, which is synchronized to *A*-engine by the data sharing model. Every transaction will include an additional logic to update the freshness table with its transaction ID, and its commit time will be recorded on the client side so that a specific transaction ID has a determined commit time. Upon starting an AP query, it scans all freshness tables to check and determine the first not-seen transaction commit time by the transaction IDs it can observe. Likewise, this method ignores the data interest of current query q but cares about the global data consistency, which may introduce much cost to care for the uninterested items.

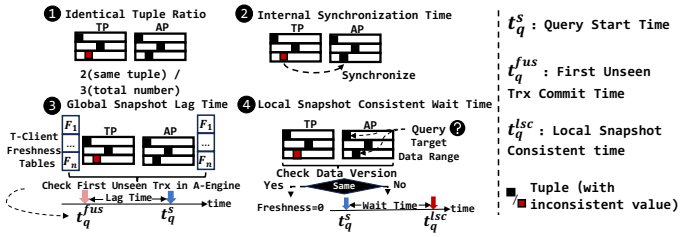


Figure 9: Data Freshness Calculation Example

4.2.1 Query-oriented Freshness. In real-world business scenarios, users care about the queried data, which should satisfy the expected freshness constraints [5, 9], instead of the global overall consistency in the whole system. None of the previous work can measure the query-oriented data freshness status. It means that even if the system is not globally consistent between *T*-engine and *A*-engine, the data queried is fresh enough for the user. This highlights the need to benchmark freshness in the granularity of query and then *Vodka* proposes to measure freshness by collecting the local snapshot consistent time duration based on the accessed data in a query q, defined in Equation. 23, where t^s_q and t^{lsc}_q are the query start time and the **l**ocal **s**napshot **c**onsistent time for q. As illustrated in Fig. 9 ❹, there exists a query where the freshness of its target data range involved will be calculated by the local snapshot consistent wait time. In a nutshell, if the data returned from the *A*-engine aligns with that from the *T*-engine, the freshness remains 0. Conversely, we will use a lightweight detection mechanism to pinpoint when they can complete the synchronization and reach a state of data local snapshot consistency. The time taken to reach this state is recorded as the freshness ($t^{lsc}_q - t^s_q$).

$$freshness(q) = max(0, t^{lsc}_q - t^s_q) \quad (23)$$

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*-engine and *A*-engine. The perfect snapshot consistency for q means the modification from L-Txn_q has been applied/synchronized to *A*-engine, 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 modification status of a row, i.e., the number of times been modified. For

instance, 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 use the last impactful transaction associated with the row's *version* and aggregate the *versions* for them in both T- and A-engine, represented by $ver_{tp}(t_q^s)$ and $ver_{ap}(t_q^s)$ respectively. T-engine synchronize new versions to A-engine, and we always have $ver_{tp}(t_q^s) \geq ver_{ap}(t_q^s)$. As proved by Theorem 6 (Detailed proof is shown in our technical report), for D_q, if the aggregated *versions* of $ver_{ap}(t_q')$ no smaller than $ver_{tp}(t_q^s)$, A-engine must cover all the latest versions at the query start time t_q^s on T-engine. We define t_q^{lsc} as the first time to have $ver_{tp}(t_q^s) \leq t_q^{lsc} \leq t_q'$.

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 $ver_{ap}^{r_j}(t) \geq ver_{tp}^{r_2}(t_q^s)$.*

4.2.2 Freshness Fail Rate. It has been declared that different businesses have their own data freshness requirements, a.k.a. *freshness threshold*. For instance, an Internet of Things scenario prioritizes data freshness within 200ms, while online gaming requires 100ms [45]. 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 real 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 $F = \frac{|Q_{fail}|}{|Q|}$.

4.2.3 Unified Measurement among 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-engine, i.e., low freshness and 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 ($Latency^u(q)$) by aligning all systems to the strict linear data sharing model, defined by Equation. 24, among which $freshness(q)$ and $latency(q)$ are the synchronization cost and the system response latency respectively.

$$Latency^u(q) = freshness(q) + latency(q) \tag{24}$$

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

4.3 Control Data Sharing Pressure

For HTAP systems, the amount of fresh data accessed by A-engines will cause different degrees of data sharing pressure. Since more accessed fresh data introduces larger synchronization costs and resource consumption in memory, network bandwidth, or CPU, there is usually a trade-off between the performance and the guarantee of freshness in HTAP systems [44]. We then propose to sketch a **workload interaction pattern** by ξ-bounded partition overlapping as defined in Definition. 1, which declares the overlapping amount of fresh data access between T- and A-engine within a freshness data bound ξ.

Definition 1. *ξ-bounded workload interaction pattern: Due to the new data from T-engines is often synchronized to A-engines in a parallel partition-oriented way [20, 52]. Suppose there are M partitions based on the partition rule, e.g., hash or range, involved by modifications from T-engines during the freshness requirement (time boundary) of ξ. ξ-bounded workload interaction pattern declares the ratio of partitions (α, $0 < \alpha \leq 1$) queried by A-engines w.r.t the M partitions modified within time range ξ, i.e., expecting to access $\alpha \cdot M$ fresh data partitions than have been modified in ξ.*

So we can quantify the data sharing pressure by specifying the workload interaction pattern with different ξ and α even having different partition rules.

5 VODKA IMPLEMENTATION

Almost all existing HTAP benchmarks are extended from the popular OLTP and OLAP benchmarks because they have been widely accepted to put into practice for many years. For example, CH-benCHmark and HTAPBench are extended from TPC-C and TPC-H. But until now, even though we have reached a consensus that a stitch of the OLTP and OLAP benchmarks can not simulate a real HTAP scenario [21, 32] and various methods have been proposed to make a semantic consistency of schema or workload, it still cannot satisfy the benchmarking requirement in fair comparison among HTAP systems *w.r.t* the distinct design philosophy in processing hybrid workloads. To make *Vodka* general for current benchmarking process, we select to construct an HTAP scenario based on TPC-C and TPC-H benchmark, by using the warehouse as the unit of scale factor (*SF*). In this section, we first introduce the benchmarking process in *Vodka* (§ 5.1), and then we give a short overview of the schema (§ 5.2) and workloads (§ 5.2.2). *Vodka* has been open-sourced in github[1].

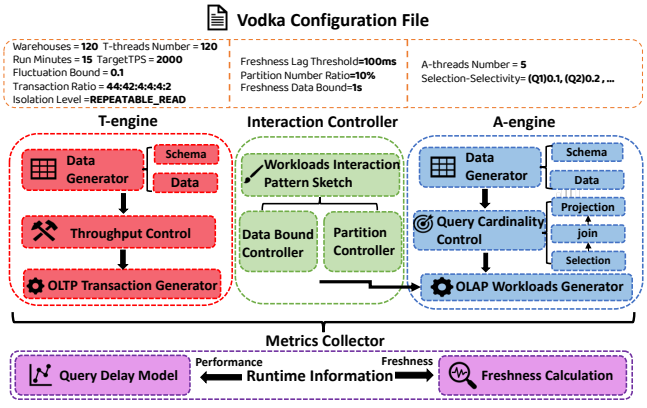


Figure 10: Vodka Architecture Overview

5.1 Design of Implementation Architecture

In Fig. 10, we show the implementation architecture of *Vodka*. The requirement of data and workload is specified in *configuration file*. Specifically, for T-engine, we determine the number of warehouses

[1] https://github.com/DBHammer/Vodka-Benchmark/

(*Warehouses*), the number of T-threads (*T-threads Number*), the target TPS (*TartgetTPS*), and the fluctuation bound for the target TPS (*Fluctuation Bound*), and so on. For A-engine, we determine the number of A-threads (*A-threads Number*) and selection-selectivity (σ-*Sel*). Based on assumptions of the inter-column independence of selection (σ-Indep) and the independence of join and selection ($\bowtie$-Indep), we have deduced that the predictability of cardinality from $\bowtie$ and σ operators by Equation 9, which is highly related to the σ-*sel* and the incremental size of reference relation (e.g. ΔR_j in Equation 15). 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 is then recorded to control the instantiation of parameters in Q1 for the stability of σ-*Sel* during the modification of T-engine as in § 3.1.1. The stable increment of reference relation can be well controlled by the throughput of T-engine as in § **??**, which then makes the projection controllable as in § 3.1.3. *Partition Number Ratio* (α) and *Freshness Data Bound* (ξ) are specified to control the interaction between hybrid workloads, i.e., determine the range of fresh data to read as in § 4.3. *Freshness Lag Threshold* ($\mathcal{T}$) in § 4.2.2 are used to quantify the freshness of reads and to calculate *freshness fail rate*.

During benchmarking, T-engine generates the workload and data following TPC-C specification but with a stable throughput. In T-engine, its schema and data are generated follow TPC-H, but we dynamically instantiate query parameters for TPC-H workload, such as the o_entry_d, $ol\_delivery\_d$, $ol\_receipt\_d$. Reservoir-based sampling is used to instantiate query parameters for TPC-H workload and guarantee σ-*Sel* for selection operators in workloads. The cardinality control of join and projection operators is highly related to throughput of T-engine. Finally, we collect the running result by Metrics Collector, which models the relationship between query latency and data size. By sketching different interaction patterns to simulate various data sharing pressures, we expose the query-oriented freshness.

5.2 HTAP Scenario Synthesis

Compared to previous work, during implementation, we have adjusted to achieve semantic consistency between TPC-C and TPC-H in both data/schema and workloads.

5.2.1 Data/Schema Consistency. For the schema, we follow CH-benchmark [13] by replacing $PartSupp$, $Part$ and $LineItem$ in TPC-H with $Stock$, $Item$ and $OrderLine$ in TPC-C, which have almost the same application semantics. All tables with the same semantics have their schema as the union of the attributes from both TPC-C and TPC-H. For the consistency of data distribution, we take the distribution rules in accordance with TPC-H. We create relation $SYNTab$s to record transaction execution times of each type of write transaction (including $New\text{-}Order$, $Payment$, and $Delivery$) in T-engine and the synchronized modifications in A-engine for calculating the real-time relation size.

5.2.2 Workload Consistency. Except for the static dimension tables such as $Country$, $Region$, and $Customer$, to guarantee semantic consistency of workload, fact tables accessed by OLAP workloads are supposed to be modified by OLTP workloads. It requires the data accessed by OLAP workloads to be also written by OLTP workloads. For example, Q12 in TPC-H accesses an attribute *ol_receipt_d*, which denotes the order receiving time of the customer. However, this field is not modified by any transaction in the original TPC-C, resulting in inconsistency between T- and A-engine for order processing and makes queries involving this field always return null for the newly generated data. To solve this problem, we introduce a new *ReceiveGoods* transaction to complete the whole life-cycle of an order after delivery as shown in Table 5. *ReceiveGoods* simulates customer actions to orders, including confirming the received orders ($ol\_receipt\_d$), rejecting the goods (ol_return_flag) and providing feedback ($o\_comment$) on the received orders. To construct such an order delivery and receiving scenario, after an order is shipped by the $Delivery$ transaction, we fill its $ol\_delivery\_d$. Subsequently, we update the $OrderLine$ relation with the $ol\_receipt\_d$ and ol_return_flag, and the $Order$ relation with the $o\_comment$.

Besides, since the real-time analysis of newly generated data by T-engine is the most compelling characteristic of HTAP systems, constructing real-time workload to benchmark data freshness is imperative for application scenario simulation as in OLxPBench [21] and HATtrick [32]. Even though OLxPBench introduces a real-time query before an OLTP write operation, it aims to reveal the impact of real-time queries on transaction latency by looking for the minimum price of goods before placing an order, while it contributes nothing to freshness calculation. In contrast, HATtrick defines a real-time query to scan all the freshness relations and provides a strict global freshness calculation between T- and A-engine, which can be too expensive for satisfying queries. Though we take a similar way to record the status of synchronization between T- and A-engine, we care about the freshness requirement of users and propose to calculate query-oriented freshness (*FreshnessCheck* in Table 5) by aggregating *access_version* in real-time query to check data consistency. Since $OrderLine$ relation meets the most frequent writes, *FreshnessCheck* is implemented to access $OrderLine$. Specifically, after a $Delivery$ transaction commits, we return the order (called *anchor order*) identified by *ol_w_id, ol_d_id, ol_o_id* and *ol_deliver_d* in $OrderLine$. Then we fill the identifier parameters for *anchor order* into the first query of *FreshnessCheck* to check data consistency between T- and A-engine by *access_version* and obtain the data access freshness (*freshness(q)*). Since database systems guarantee monotonic-read consistency [56], if the *anchor order* has been synchronized, the delivered items before it will also be visible to A-engine. 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, which collects recent orders (specified by ξ) delivered until the time of *anchor order*. Notice that the second query uses the unique *ol_deliver_d* attribute to retrieve fresh data within a specified freshness data scan boundary ξ, and ol_w_id is taken to control the size of warehouses accessed, i.e., α. Because modifications in TPC-C are uniformly distributed across warehouses, we can randomly pick a proportion of α warehouses to guarantee the size of access partitions and construct workloads to access various data

Table 5: New Workloads In Vodka

Workload Type	Name	SQL Structure
Transaction	ReceiveGoods	**BEGIN** **SELECT** ol_delivery_d **FROM** OrderLine **WHERE** ol_w_id = ? **AND** ol_d_id = ? **AND** ol_o_id = ? **UPDATE** OrderLine **SET** ol_receipt_d = ?, 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_commit_d, ol_delivery_d, ol_receipt_d **FROM** OrderLine **WHERE** ol_w_id = ? **AND** ol_d_id = ? **AND** ol_o_id = ? **END**
Real-time Workload	FreshnessCheck	**SELECT SUM**(access_version),ol_delivery_d **FROM** OrderLine **WHERE** ol_w_id = ? **AND** ol_d_id = ? **AND** ol_o_id = ? **SELECT** * **FROM** OrderLine **WHERE** ol_w_id >= ? **AND** ol_w_id <= ? **AND** ol_delivery_d **In** (?,?,?,...)

sizes for *latency(q)*. In practice, according to ξ, we maintain a fixed time range of the latest *ol_deliver_d* in a circular array, which is updated once a *Delivery* transaction is committed. For the second query in *FreshnessCheck*, we locate the delivery data of each order that falls in the circular array delivered before the *ol_deliver_d* of the *anchor order*. Notice that following the processing logic of TPC-C workload under a predefined TPS of *T*-engine, it is feasible to calculate the size of the array to store delivered order date by the time range of ξ, which are all inserted as a candidate of the 'IN' parameter, details in our technical report.

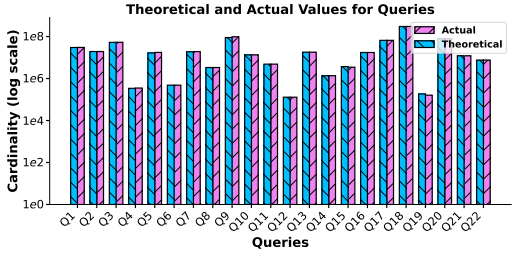


Figure 11: Cardinality Relative Errors of Queries

6 EXPERIMENTAL EVALUATION

In this section, we run extensive experiments to illustrate the effectiveness of *Vodka* and explore the HTAP performance and freshness of three kinds of open-source HTAP database systems, which are O PostgreSQL Streaming Replication(PostgreSQL-SR) [41], ceanBase [59] and TiDB [20].

6.1 Experimental Configuration

Before delving into the concrete experiments, we first describe the experimental configurations in cluster environment, the setup for each chosen database system and the default configurations.

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 sever for OLTP and OLAP tasks simultaneously. We deploy an additional proxy called OBProxy (v4.2.0) to dispatch OLTP and OLAP tasks for it. For TiDB (v7.1.0), its row store called TiKV and column store called TiFlash are deployed into each server by manually binding to CPU cores in half [10, 31], allowing each node to handle both OLTP and OLAP tasks simultaneously. We deploy one HAProxy (v2.6.2) for OLTP tasks and one HAProxy for OLAP tasks in TiDB. PostgreSQL-SR (v14.5) is designed based on a 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 and the default synchronization time interval '*recovery_min_apply_delay*'=0s (second).

Benchmark configuration. By default, all experiments are performed on a data size of approximately 8GB, i.e., 120 warehouses and 120 *T*-threads, where each thread is binded with one warehouse [44, 61]. We set each system to have 16 query parallel degree, which is the number of concurrent queries. The default reservoir sampling size is 40*K* rows (approximately 1MB) according to Hoeffding's Inequality with its theoretical error bound $\delta = 1\%$ in a confidence level of $\hat{\alpha} = 99.9\%$ to determine σ-*Sel* of selection operator. During benchmarking, to avoid code-start problem, all experiments have 5-minute warm-up phase. To have a fair comparison, we run all experiments under the optimal OLAP performance while guaranteeing the same stable OLTP throughput.

6.2 Evaluation of the Key Design of *Vodka*

In this section, we first conduct preliminary experiments for determining some experimental configuration parameters (§ 6.2.1). Then we launch experiments to verify *Vodka*'s key design, especially in the control of cardinality stability (§ 6.2.2), the fidelity of model fitting (§ 6.2.3) and the controllability in OLTP throughput(§ 6.2.4).

6.2.1 Preliminary Experiments on Parameter Configuration. We first conduct an experiment to discover the maximum OLTP throughput for three systems under 120 *T*-threads by employing one *A*-thread, which has the minimized impact on performance of *T*-engine. It is used to determine the maximum throughput threshold of *T*-engine for all three HTAP systems to achieve a stable and consistent increasing of data. The results reveal that PostgreSQL-SR achieves the highest TPS of 6.5×10^3 due to its standalone nature, that is writing on a single node without any distributed transactions; OceanBase has the TPS of 5×10^3, benefiting from its continuous optimizations in LSM-Tree-based storage design and Paxos-based 2PC implementation [60]; TiDB has a relative low TPS of 2×10^3. So we set the TPS no larger than 2×10^3 in default. To guarantee TPS=2×10^3, we find the maximum number of *A*-threads is 10 for all the three systems. So we set *A*-threads no larger than 10 for the subsequent experiments. In default we set TPS=2×10^3 and *A*-threads=10.

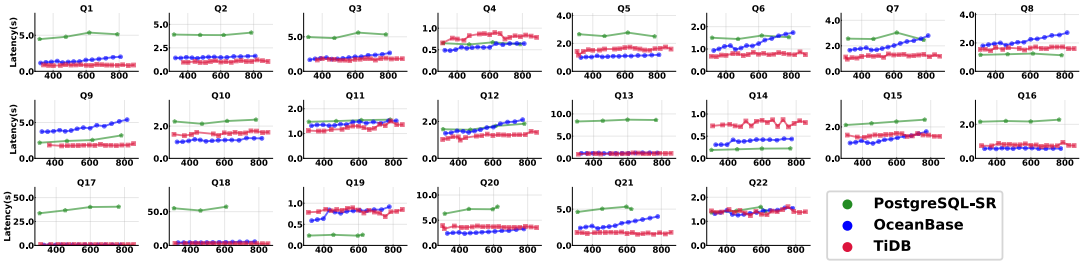


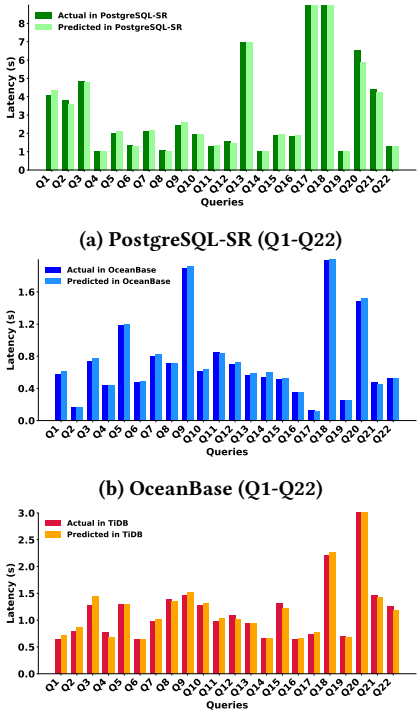
Figure 12: Query Latency *vs.* Data Size

6.2.2 Cardinality Stability of OLAP Queries. We adopt *relative error*= $\frac{1}{|Q_{op}|}\sum_i \frac{||C_i|-|\hat{C}_i||}{|C_i|}$ [28] to measure the accuracy of cardinality control to query Q by all its operators ($|Q_{op}|$), where $|C_i|$ and $|\hat{C}_i|$ are the runtime real cardinality and the theoretical cardinality of the i^{th} operator at a specific time point. 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 query templates, we take PostgreSQL-SR to illustrate *relative error* of our cardinality estimation by running OLTP workload with the default 2×10^3 TPS for 20m (minutes) and then run the whole OLAP workload. The result in Fig. 11 demonstrates all queries have almost no relative errors (the maximum is 1.7% of Q19).

Meanwhile, we keep running TPC-C workloads and demonstrate OLAP query latency in Fig. 12 for Q1-Q22 by setting TPS=2×10^3 for T-engine and 10 A-threads. As the increasing of test time (database size), all query latencies expose linear changes. It verifes that a linear model can be sufficient for fitting the data.

It is interesting to find that these HTAP systems have performed differently for queries. PostgreSQL-SR performs better than OceanBase and TiDB for a small number of queries, i.e., Q8, Q14, Q19 and Q22, which benefit from the well implementation of the Bitmap Index Scan (reduce scan cost). Besides, OceanBase demonstrates an overall better performance, especially for Q5, Q10, and Q20, which benefit from its optimization in join order selection and performing nested-loop-join in advance for small tables to reduce the size of intermediate result sets and reduce join cardinalities.

6.2.3 Effectiveness of Linear Modeling to Latency. Under the configuration of TPS=2×10^3 and 10 A-threads, we utilize all three systems to validate the effectiveness of the latency modeling function $L(S) = k \cdot S + b$. We try to learn parameters k and b for each query based on 10 minutes performance results excluding the first 5-minutes warmup. We gather 100 pairs of the latency with the calculated data size every 6 seconds based on throughput of T-engine to organize the training set. We then decide θ from the candidate set {0.01,0.1,1,10,100}, which has been adopted by previous work [25], to fit parameters k and b by running an extra 5 minutes 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 and θ=0.1 performs the best. Then we continue running another 5 minutes and sample the latency with the calculated data size every 6 seconds and have 50 pairs of test test. We compare the average actual latency with the average predicted latency in Fig. 13, which shows the maximum performance deviation < 5% in each system by our linear modeling.



(a) PostgreSQL-SR (Q1-Q22)

(b) OceanBase (Q1-Q22)

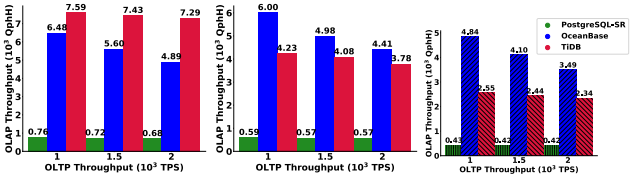
(c) TiDB (Q1-Q22)

Figure 13: Actual and Predicted Latency For Queries

6.2.4 Controllability in OLTP throughput. In Table. 6, we take the *TPS deviation*=$\frac{|TPS_t - TPS_a|}{TPS_t}$ to demonstrate the deviation between the target and actual throughputs (TPS_t and TPS_a) by our throughput control method. We observe that the average TPS deviation of all systems under different target throughputs (1×10^3, 1.5×10^3 and 2×10^3) is less than 1%. It indicates that the target throughput can be guaranteed effectively, which is the prerequisite for stable data size increasement.

Table 6: Relative Deviation in Controlling OLTP Throughput

System/Relative Deviation	Throughput		
	1K	1.5K	2K
OceanBase	0.1%	0.1%	0.1%
TiDB	0.1%	0.1%	0.3%
PostgreSQL	0.1%	0.1%	0.1%



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

Figure 14: Performance of A-engine under Different TPS

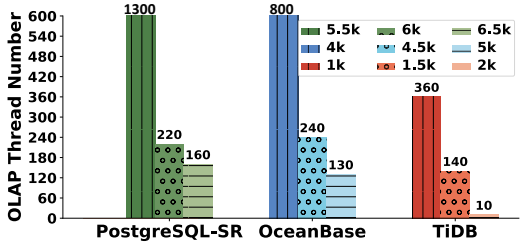


Figure 15: The Maximum OLAP Threads Number

6.3 Comparison of HTAP Systems for Resource Isolation and Data Sharing

In this section, we compare the effect of the key techniques in different HTAP systems. We benchmark resource isolation in § 6.3.1 and data sharing in § 6.3.2.

6.3.1 Resource Isolation of HTAP Systems. We first demonstrate the OLAP performance (QphH) under different TPS by changing *A*-threads in Fig. 14. For example in Fig. 14a, the OLAP performance of TiDB conquers the other two systems. When TPS=2×10^3 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) [39], e.g., column store, compression, vectorization execution and operation optimizations, etc. 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 performance of *A*-engine is inferior to the other two systems. We have observed an interesting phenomenon that as throughput increases, OceanBase experiences a relatively evident drop in single *A*-thread QphH. This might be due to OceanBase's MVCC mechanism facing longer version chains when write volumes rise, impacting OLAP scan efficiency. In contrast, TiDB, with its separate columnar replica and partition pruning in TiFlash, shows less performance sensitivity to data size changes. PostgreSQL-SR, due to its excellent bitmap index design, exhibits less noticeable performance degradation as data size increases. Additionally, when increasing *A*-threads, all systems have per-thread QphH reduction, with TiDB being the most affected. This is primarily due to multithreading contends and consumes a lot of system resources in parallel, especially in CPU resources, resulting in interference and performance degradation among them. Therefore, we can observe noticeable performance drop in TiDB. Moreover, we find an optimization technique in OceanBase known as 'small table broadcasting'. This technique is primarily used to replicate tables with low update frequencies, such as those in TPC-H (e.g., *Nation*, *Region*, *Item*), across all servers. By this, queries accessing these tables are localized, leading to more efficient execution strategy generation. In our experiments under 10*A*-threads, we conduct additional evaluations of OceanBase's OLAP performance under three different throughputs. We observe that disabling small table broadcasting led to a QphH reduction ranging from 3% to 7%. For HTAP systems, especially in a *decoupled-style* architecture, optimizing tables with infrequent updates through small table broadcasting seems to be a valuable consideration.

Since each system has a different throughput frontier by the given *T*-threads, resource contentions from *A*-engine vary under different OLAP pressure 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 resource 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 § 6.2.1, among which PostgreSQL-SR performs the best, i.e., TPS frontier is 6.5×10^3 with 1 *A*-thread. The maximum *A*-threads number is shown in Fig. 15. Specifically, we monitor the CPU, memory, network IO, and disk IO during benchmarking. We collect the resource usage for PostgreSQL-SR (6.5×10^3), OceanBase (5×10^3), and TiDB (2×10^3) under their maximum supported *A*-threads in Table 7. In Fig. 15, we find that PostgreSQL-SR demonstrate the best resource isolation effects since its maximum supported *A*-threads is more than the others. For instance, when TPS reaches 5.5×10^3, PostgreSQL-SR can effectively accommodate nearly 1300 concurrent *A*-threads. In contrast, OceanBase can support fewer *A*-threads, approximately 130, under TPS = 5×10^3 lower than 5.5×10^3. This difference arises from the fact that OceanBase's resource isolation on a single node is not as robust as the resource isolation between nodes in PostgreSQL-SR. On the other hand, TiDB faces challenges in supporting relatively low TPS scenarios, primarily due to its substantial CPU utilization and network bandwidth consumption. We use the following resource utilization results in Table. 7 to make a further explanation. Except for PostgreSQL-SR, which employs separate write (W-Node) and read (R-Node) nodes, the other systems utilize combined write and read nodes (W&R-Node) on each node. The other TPS scenario has the same resource consumption features.

Table 7: Resource Consumption Under Respective Maximum Supported *A*-threads

System/Resource	PostgreSQL-SR W-Node	PostgreSQL-SR R-Node	OceanBase W&R-Node	TiDB W&R-Node
CPU Utilization	20%	**60%**	**80%**	**80%**
Memory Utilization	60%	60%	60%	30%
NetWork IO (kbps)	4E+05	2E+05	1E+04	**5E+05**
Dirty Page Number	1E+05	**4E+06**	1E+04	5E+04

We find that the final performance bottleneck is concentrated on the CPU, which generally reaches more than 60% utilization. On the slave read node of PostgreSQL-SR, increasing the number of *A*-threads can lead to a continuous rise in the number of dirty pages (4E+06 dirty page number in R-Node). This is because PostgreSQL-SR adopts the Copy-on-Write (COW) mechanism, which requires

maintaining multiple historical data versions, potentially causing difficulties in flushing data to disk and impacting throughput. For OceanBase, its network bandwidth and memory consumption are relatively low, but CPU resources are consumed more, and it will end up close to 80% utilization, which is higher than the CPU utilization of PostgreSQL-SR in the performance bottleneck state. For TiDB, even though it deploys TiKV and TiFlash separately, it needs to confirm the consistency of the OLAP query version and OLTP through frequent network communication, so the consumption of network resources is more significant (5E+05 kbps in W&R-Node).

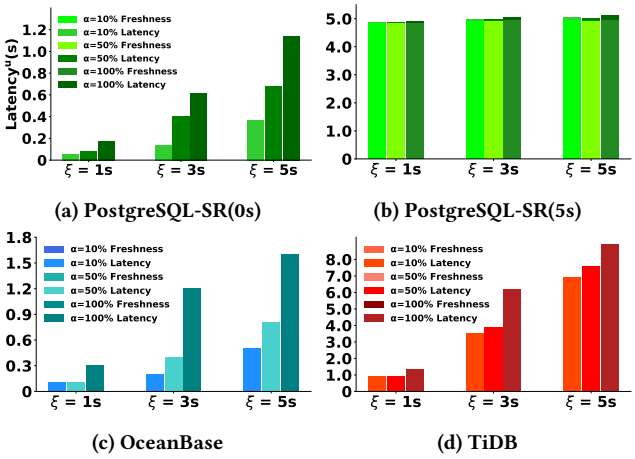
6.3.2 Freshness Comparison. In this experiment, we set TPS=2 × 10^3, and control the workload interaction pattern by selecting ξ from 1, 3, 5 seconds and α from 10%, 50%, 100% respectively to simulate data sharing pressure. We select four common scenarios with varying freshness requirements to assess the impact of data sharing models by freshness lag time $\mathcal{T}$. 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) [45]. When accessing fresh data across all warehouses (α=100%), approximately 8,000 rows of new data can be generated per second. We run a hundred rounds of the real-time workload *FreshCheck* randomly and observe that the *freshness error rate* of these three systems remains the same for all combinations of α and ξ, as shown in Table. 8. PostgreSQL-SR is a general master-slave architecture by transferring the WAL logs from primary to the target slave server asynchronously. Its freshness depends on the pre-assigned synchronization interval. Specifically, it provides a parameter *'recovery_min_apply_delay'* (=0 in default) to control the minimum latency to synchronize logs to slave servers. When this parameter is set to 0, even in asynchronous synchronization replication, PostgreSQL-SR's synchronization speed can meet the real-time business needs under current TPS. However, when the parameter is set larger to 5s, it can no longer support (near) real-time business requirements. In contrast, OceanBase and TiDB have shown to maintain a *freshness fail rate* of 0. Specifically, OceanBase is similar to a traditional relational database management system (RDBMS) but it is distributed, which 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. TiFlash acts as a learner within the raft group, receiving and organizing data without participating in leader voting. When a new query arrives, it sends a request to its leader to obtain the latest data and synchronizes it by replaying logs. This results in a consistent freshness fail rate of 0.

Furthermore, we demonstrate the *unified linearly- consistent latency* ($Latency^u$) in Fig. 16, 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 § 4.2.3. We collect $Latency^u(q)$ for the real-time workload FreshnessCheck. First, for PostgreSQL-SR, when *'recovery_min_apply_delay'*= 0s, its overall latency remains relatively low as we increase the number of warehouses α and freshness data bound ξ. This is mainly due to its effective log synchronization of small data size and log replaying mechanisms, as well as its bitmap index scan. However, when we set larger synchronization intervals, i.e., *'recovery_min_apply_delay'*=5s, the overall latency is dominated by data synchronization. For OceanBase and TiDB, when we increase α and ξ, there is also a noticeable increase in overall latency. OceanBase does not experience a delay of data synchronization from *T*-engine to *A*-engine. Its B+tree index allows for a quick data location, resulting in lower overall latency. In contrast, Synchronizing more data in TiDB leads to obvious synchronization delays, and when the 'IN' condition of the FreshnessCheck query includes many items, i.e., larger α, it starts a full table scan in default, causing a higher execution cost.

Table 8: Freshness Fail Rate in Various Businesse Scenarios

Systems/Business	Fraud Detection[1]	Online Gaming[2]	IOT System Monitor[3]	Online E-Commerce[4]
OceanBase	0%	0%	0%	0%
TiDB	0%	0%	0%	0%
PostgreSQL-SR(0s)	0%	0%	0%	0%
PostgreSQL-SR(5s)	100%	100%	100%	0%

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



(a) PostgreSQL-SR(0s) (b) PostgreSQL-SR(5s)

(c) OceanBase (d) TiDB

Figure 16: Real-time Queries's Freshness and Latency

In order to make the results in three HTAP systems comparable, we have set TPS=2K. Since a low TPS generates a slow increasing of data, a small data synchronization in PostgreSQL-SR has no impact on performance. We then increase TPS (=5 × 10^3) 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 α = 100% and the fresh data bound ξ = 5*s*. We run the other 100 rounds of FreshCheck workloads and collect the information of freshness latency in Table. 9. We observe that OceanBase always achieves the highest freshness with *fresh error 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. Specifically, we find that PostgreSQL-SR's *freshness fail rate* is 80%, 40%, 40%, and 0% respectively.

Table 9: 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

7 CONCLUSION

This paper proposes a comprehensive benchmarking methodology for HTAP systems and implements a benchmark called Vodka on the basis of TPC-C and TPC-H. We introduce two theoretical proofs in hybrid workloads complexity cardinality stability control and HTAP architecture classification from the perspective of data consistency. We also propose the method of performance measurement by fitting latency and data size to acquire corresponding performance under any data size and the calculation method of query-oriented data freshness with the control of workload interaction pattern. Evaluations demonstrate that we can maintain stable hybrid workload cardinality control by regulating OLTP throughput and OLAP workload selectivity. Moreover, we can show a fair and comparable result comparison in HTAP key techniques of resource isolation and data sharing.

8 ACKNOWLEDGEMENTS

This work was supported by Ant Group Research Fund.

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, 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. 1817–1819.

[4] Daniel G Bobrow, Jerry D Burchfiel, Daniel L Murphy, and Raymond S Tomlinson. 1972. TENEX, a paged time sharing system for the PDP-10. Commun. ACM 15, 3 (1972), 135–143.

[5] Mokrane Bouzeghoub. 2004. A framework for analysis of data freshness. In Proceedings of the 2004 international workshop on Information quality in information systems. 59–67.

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

[7] 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.

[8] 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

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

[10] 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, 1327–1329.

[11] 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. 169–182.

[12] 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. 293–304.

[13] 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. 1–6.

[14] Jim Diederich and Jack Milton. 1988. New methods and fast algorithms for database normalization. ACM Transactions on Database Systems (TODS) 13, 3 (1988), 339–365.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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. arXiv preprint arXiv:2203.16095 (2022).

[22] 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, 195–206.

[23] 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. 49–64.

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

[25] 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, 11844–11867.

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

[27] 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.

[28] 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). 575–586.

[29] Lo, Eric, Cheng, Nick, Choi, Byron, Hon, Wing-Kai, Lin, and Wilfred. 2014. MyBenchmark: generating databases for query workloads. Vldb Journal the International Journal of Very Large Data Bases (2014).

[30] 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. 2530–2542.

[31] 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. 37–50.

[32] 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. 1810–1824.

[33] 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. 499–510.

[34] Patrick E O'Neil, Elizabeth J O'Neil, and Xuedong Chen. 2007. The star schema benchmark (SSB). Pat 200, 0 (2007), 50.
[35] Thorsten Papenbrock and Felix Naumann. 2017. Data-driven Schema Normalization.. In EDBT, Vol. 17. 342–353.
[36] Andrew Pavlo and Matthew Aslett. 2016. What's Really New with NewSQL? SIGMOD Rec. 45, 2 (sep 2016), 45–55. https://doi.org/10.1145/3003665.3003674
[37] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. In 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10).
[38] 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..
[39] PingCAP. 2023. TiFlash Overview. https://docs.pingcap.com/tidb/v7.2/tiflash-overview..
[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] Adam Prout, Szu-Po Wang, Joseph Victor, Zhou Sun, Yongzhu Li, Jack Chen, Evan Bergeron, Eric Hanson, Robert Walzer, Rodrigo Gomes, et al. 2022. Cloud-Native Transactions and Analytics in SingleStore. In Proceedings of the 2022 International Conference on Management of Data. 2340–2352.
[43] 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.
[44] 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. 2043–2054.
[45] Sijie Shen, Rong Chen, Haibo Chen, and Binyu Zang. 2021. Retrofitting High Availability Mechanism to Tame Hybrid Transaction/Analytical Processing. In 15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21). 219–238.
[46] Utku Sirin, Sandhya Dwarkadas, and Anastasia Ailamaki. 2021. Performance characterization of HTAP workloads. In 2021 IEEE 37th International Conference on Data Engineering (ICDE). IEEE, 1829–1834.
[47] TATP. 2009. TATP Benchmark. https://tatpbenchmark.sourceforge.net/.
[48] TPC. 1992. TPC-C Benchmark. http://www.tpc.org/tpcc/.
[49] TPC. 1999. TPC-DS Benchmark. http://www.tpc.org/tpcds/.
[50] TPC. 1999. TPC-H Benchmark. http://www.tpc.org/tpch/.
[51] TPC. 2011. TPC-E Benchmark. http://www.tpc.org/tpce/.
[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. 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] 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. 94–107.
[55] Christian Winter, Moritz Sichert, Altan Birler, Thomas Neumann, and Alfons Kemper. 2023. Communication-Optimal Parallel Reservoir Sampling. BTW 2023 (2023).
[56] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. 2019. Autoscaling tiered cloud storage in Anna. Proceedings of the VLDB Endowment 12, 6 (2019), 624–638.
[57] 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.
[58] 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.
[59] 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.
[60] 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.
[61] 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. 95–98.

Received 20 February 2023; revised 12 March 2023; accepted 5 June 2023