How Good is My HTAP System?

Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, Xiangyao Yu
University of Wisconsin-Madison
{milkai,kpgaffney}@wisc.edu,{chronis,zhihan,jignesh,yxy}@cs.wisc.edu

ABSTRACT

Hybrid Transactional and Analytical Processing (HTAP) systems have recently gained popularity as they combine OLAP and OLTP processing to reduce administrative and synchronization costs between dedicated systems. However, there is no precise characterization of the features that distinguish a good HTAP system from a poor one. In this paper, we seek to solve this problem from the perspectives of both *performance* and *freshness*. To simultaneously capture the performance of both transactional and analytical processing, we introduce a new concept called *throughput frontier*, which visualizes both transactional and analytical throughput in a single 2D graph. The throughput frontier can capture information regarding the performance of each engine, the interference between the two engines, and various system design decisions. To capture how well an HTAP system supports real-time analytics, we define a *freshness* metric which quantifies how recent is the snapshot of the data seen by each analytical query. We also develop a practical way to measure freshness in a real system. We design a new hybrid benchmark called *HATtrick* which incorporates both *throughput frontier* and *freshness* as metrics. Using the benchmark, we evaluate three representative HTAP systems under various data size and system configurations and demonstrate how the metrics reveal important system characteristics and performance information.

CCS CONCEPTS

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

KEYWORDS

HTAP; throughput frontier; freshness score; benchmark

ACM Reference Format:
Elena Milkai, Yannis Chronis, Kevin P. Gaffney, Zhihan Guo, Jignesh M. Patel, Xiangyao Yu. 2022. How Good is My HTAP System?. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526148

1 INTRODUCTION

Modern enterprises often have both *online transactional* (OLTP) and *online analytical processing* (OLAP) workloads, and they typically serve these workloads using separate databases. Maintaining multiple database instances, however, requires additional administrative effort and computational resources. In recent years, there is a growing demand for database systems that can serve both OLTP and OLAP workloads with performance comparable to more specialized systems. This approach, called *Hybrid Transactional and Analytical Processing* (HTAP), is now an active area of research.

Interestingly, even after a hugefl urry of papers in the HTAP field [4–6, 10, 11, 13, 15, 17, 18, 20–23, 32, 33, 37, 38, 43], there is no clear definition of what makes an HTAP system *good*, beyond its ability to support a wide range of mixed OLTP/OLAP workloads. As a result, it is difficult to characterize and compare the performance of HTAP systems. To address this limitation, we define a systematic approach to characterize the behavior of HTAP systems.

We propose two methods that define and measure HTAP performance. First, a method to characterize an HTAP system based on a performance-centric definition. Second, a method to measure the ability of an HTAP system to provide real-time analytics.

In practice, we rarely see pure OLTP or OLAP workloads; for example, TPC-E [1] has a mix of both OLTP and OLAP queries. Nevertheless, we build on these two extreme cases (pure OLTP and OLAP) and call the space in between the HTAP spectrum. Conceptually, for any point on the HTAP spectrum, a good HTAP system does not favor the OLTP or OLAP component at the expense of the other component. We capture the overall performance over the HTAP spectrum with a new concept called *throughput frontier*. This concept combines the performance of the OLTP and OLAP workload components. By visualizing the throughput frontier in a 2D chart, we can understand the global performance behavior of an HTAP system, as well as identify problematic areas.

A key concept in the HTAP domain is *freshness*. Loosely defined, the freshness of an HTAP system is a measure of delay with which updates to the database (from the OLTP component of the workload) are made visible to the OLAP queries. A more precise definition of freshness will be provided in Section 4.1. We also propose a method to measure *freshness* in practice for any HTAP system. The empirical value of *freshness* allows us to understand whether the HTAP system is able to provide real or near-real time analytics.

We present a new benchmark called *HATtrick*, which we use to validate the concept of throughput frontier and our proposed method to measure freshness. The HATtrick benchmark employs a family of systematically generated workloads. At one extreme is a purely transactional workload, and at the other extreme is a purely analytical workload. In between, the workload has a mix of transactional and analytical operations. For each workload, HATtrick measures and extracts the performance and freshness of the HTAP system at that "operating point". To run HATtrick, one executes a range of workloads as specified by the benchmark parameters.

Finally, we run the HATtrick benchmark on a number of database systems to understand their characteristics. Our results show that the shape of the throughput frontier reveals essential information about the ability of a database system to concurrently serve

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on thefi rst page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526148

OLTP and OLAP workloads. Also, it provides information about the way the system scales in different mixes of hybrid workloads and how it shares its resources amongst the two workload components. Moreover, our freshness measurement method reveals the ability of the system to provide the latest updates to the analytical queries. Results show that there is still room for improvement for database systems to become more performant in the HTAP space.

In summary, this paper makes the following contributions: we introduce the *throughput frontier* in Section 3. Then, we define *freshness* and introduce a practical way to measure it in Section 4. Then, we design a benchmark named *HATtrick* in Section 5 and use it to evaluate and analyze three database systems in Section 6. Collectively, we provide a more principled approach to characterize and evaluate HTAP systems.

2 MOTIVATION

In this section, wefi rst present a classification of HTAP systems based on their *performance isolation* and *freshness* properties. Then, we describe existing work on benchmarking HTAP systems. We then motivate the need for our proposed *HATtrick* benchmark.

2.1 Design challenges

Generally speaking, an HTAP system should achieve the following two goals: (i) *performance isolation* — the transactional and analytical workloads should not interfere with each other, and (ii) *freshness* — analytical queries should observe the latest transactions' updates.

An H**TA**P database contains two workloads, an OL**T**P workload and an OL**A**P workload, against the same physical database. For simplicity, we refer to these two workloads as the T and the A workloads. T workloads typically include a mix of read and write transactions, each of which operates on a small subset of the database and uses indexes to accelerate search. In contrast, A workloads are mostly read-only and often involve scans, joins, and aggregates of large subsets of the database.

An HTAP system achieves ideal performance isolation when each of the T and A workloads achieves the performance as if it was executed independently. This is a desirable behavior, since it allows the two workloads to run without one blocking or affecting the performance of the other. Therefore, the practical challenge is to design a system that can share the resources between the two workloads in a way that minimizes the interference between them.

Moreover, an HTAP system should allow every A query to read the latest modifications of the T workload. These modifications produce *fresh* data. We say that an HTAP system achieves perfect *freshness* when there is no delay from the time the T workload commits its changes to the time the A workload is able to process the same data. The challenge is to provide freshness without negatively impacting the performance of the T or the A workloads.

In the next section, we describe how different HTAP designs use different solutions to achieve performance isolation and freshness.

2.2 Design classification

HTAP systems today follow many different designs. We classify them based on their architectures into three categories: (i) *shared design*, (ii) *isolated design*, and (iii) *hybrid design*. Then, we provide some representative examples in each category.

Shared design: Systems that belong to this category execute the T and A workloads in a single engine. They maintain a single copy of data and share resources between the two workloads (e.g., memory bandwidth, CPU cores, and shared caches). Examples of systems that belong to this category include all the traditional relational databases such as PostgreSQL [28, 39, 40], DB2 [16], and Oracle [24] but also specialized in-memory databases such as SAP HANA [10, 37], Hyper [17, 23], and DB2 BLU [32]. Systems that follow the shared design use various ways to provide isolation between the two workloads. Creating *snapshots* of the main database is one way to create "data replicas" and reduce the interference between reads and writes. So, they use the copy-on-write (CoW) or multiversion concurrency control (MVCC) mechanisms. Each of the systems that we mention above use their snapshot isolation mechanism to achieve fresh analytics. For example, in MVCC every analytical query that arrives needs to traverse potentially lengthy version chains [41] andfi nd the right snapshot.

Isolated design: Systems that belong to this category usually provide compute isolation and dedicated resources to each workload. This is achieved by using different NUMA nodes for each workload or even different machines. Also, two different copies of the data are maintained, which have different representations. For example, row-store format is used in the T engine and column-store format is used for the A engine, which supports efficient data compression for processing high volumes of data in-memory. Examples of systems that belong to this category are BatchDB [21], TiDB [15], SAP HANA SOE's [13], F1 Lightning [43], Wildfire [6], Db2 event store [11], Greenplum [20], PostgreSQL Streaming Replication [27], and the fractured mirrors [8]. An advantage of the systems that follow the isolated design is the mitigation of the interference between the two workloads since there is no sharing of resources. For achieving fresh analytics, the systems above traditionally follow an ETL process. Recent solutions aim to more frequently update the A replica of the data and achieve higher freshness.

Hybrid design: This category combines characteristics from the two previously mentioned designs. Systems that belong to this category usually are in-memory databases which execute the two workloads in a single machine with shared resources but maintain two copies of data with different representations. Examples of systems that belong to this category are Microsoft SQL Server with Hekaton [9, 19], Oracle dual-format DB [18], and SingleStore [38]. Maintaining two copies of the data is a way for these systems to aim for performance isolation. To provide fresh analytics, every analytical query before execution has to fetch the changes from the transactional log or the tail of the T copy.

2.3 Current HTAP benchmarks

Existing popular HTAP benchmarks include CH-Benchmark[31], HTAPBench [7], and Swarm64 [29]. We identify important limitations in the current HTAP benchmarks. For each limitation discussed below we briefly discuss the strategy we will follow in HATtrick, the benchmark proposed in this paper.

Unable to measure performance isolation. The current hybrid benchmarks cannot identify whether a tested system is achieving performance isolation between the T and the A workloads. HTAPBench and Swarm64 view one of the workloads as the primary,

usually the T, and the other as a turbulence of the primary. Their goal is to execute the secondary workload without affecting the target throughput of the primary workload. In HATtrick we view the T and A workloads as equal. Our primary goal is to discover how good the current HTAP systems are at achieving performance isolation when both workloads are equal. The *throughput frontier* metric, which we will discuss in detail in Section 3.1, shows how close is a system at achieving performance isolation, how performance scales, and the interference of the two workloads.

Unable to measure freshness. The second limitation of existing benchmarks is that they cannot measure the freshness of an HTAP system. CH-Benchmark is the only benchmark that identifies freshness as an important factor in system performance. They show how the performance is affected by different freshness configurations in the old version of the Hyper [17] database. However, they do not provide any methodology for measuring the freshness of a system. In HATtrick, we provide a method to measure freshness applicable to all HTAP design categories discussed in Section 2.2. Our method is simple and can be adopted by any HTAP benchmark with minimal changes, we provide more details in Section 4.

Unable to identify design category. In Section 2.2 we categorize HTAP systems based on their architectures. These categories have also been discussed in other research works [12, 14, 25, 33] and they are important to understand and improve an HTAP system. None of the current hybrid benchmarks is able to discover the category of a tested system. HATtrick can extract this information and communicate it to the user in a friendly way. Our evaluation in Section 6 will show how HATtrick discovers the correct category.

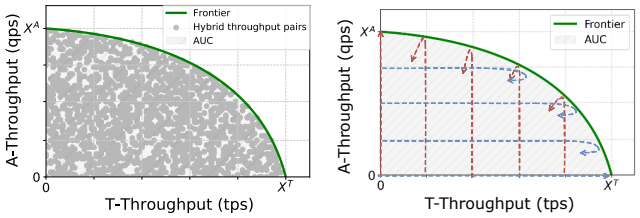
Complicated schemas. Existing benchmarks are created by combining the schemas of TPC-C [3] and TPC-H [2]. The TPC schemas are complex which makes their implementation not straightforward to the users. In the world of OLAP, this has led to the creation of the SSB [26] benchmark which is based on TPC-H but significantly simplified. SSB follows the Kimball [35] definition of a data warehouse, based on which the data model should have a star schema and is widely used due to its simplicity. Compared to TPC-H, the SSB schema allows more efficient table and column compression and eliminates the restrictions in partitioning and indexing [36]. We believe that HTAP systems can benefit from the above schema improvements. In our proposed benchmark we extend the SSB schema to support a new T workload which is an adapted version of TPC-C. We discuss the design of HATtrick in Section 5.

Hard to compare multiple HTAP systems. Although existing benchmarks can be used to compare multiple HTAP systems, they do not provide a systematic way for achieving it. We focus on combining all the information needed to compare different HTAP systems into a small set of metrics. We also provide a visualization of the metrics to make the comparison process more intuitive.

Due to the above limitations, we believe that there is still space for research in benchmarking HTAP systems and this work is a step towardsfi lling this gap with the proposed HATtrick benchmark.

3 PERFORMANCE-CENTRIC DEFINITION OF HTAP SYSTEMS

Although many HTAP systems with different designs exist, it is not clear how their performance should be measured and compared with each other. In this section, we introduce the concept of the *throughput frontier* and define the performance characteristics that capture the key properties of an HTAP system.



(a) Sampling method. **(b)** Saturation method.

Figure 1: An illustration of throughput frontier and different methods of creation.

3.1 Throughput frontier

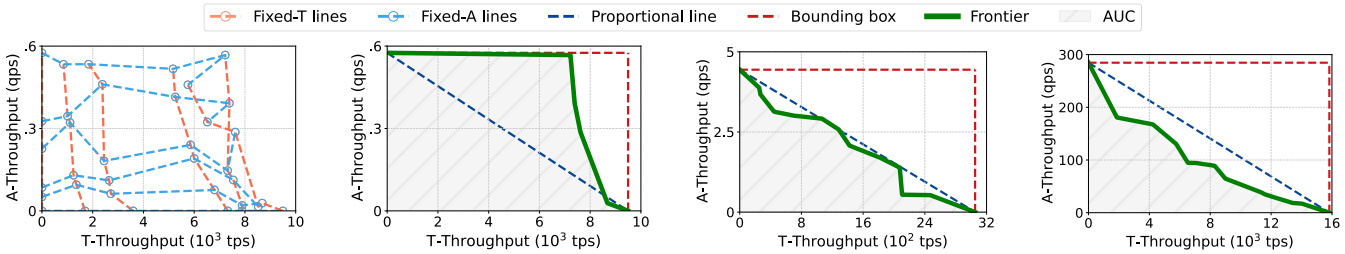
The performance of an OLTP or OLAP system is typically characterized by plotting *throughput* versus the number of clients. However, characterizing HTAP performance is more complex.

We consider a hypothetical HTAP system that serves a mix of T- and A-clients, each of which issues a constant stream of requests. We model the performance of the system using a function S. The input to S is a 2-tuple $(\tau,\alpha) \in \mathbb{N}^2$, where τ and α are the number of T- and A-clients, respectively. The performance of S is a 2-tuple $(x_t, x_a) \in \mathbb{R}^2_{\geq 0}$ where x_t and x_a are the T- and A-throughputs, respectively. We refer to the 2-tuple (x_t, x_a) as the *hybrid throughput* of S.

Fortunately, we can make the simplifying assumption that S is bounded. Then, the most interesting set of points for HTAP performance characterization are those in the bound. Intuitively, these points represent the maximum *hybrid throughput* that can be achieved by the system across all configurations of clients. Of course, real HTAP systems cannot be perfectly modeled as described above. However, as our experiments demonstrate, it is possible to estimate a reasonably smooth curve that denotes a system's maximum achievable hybrid throughput. For the remainder of this paper, we refer to this curve as the *throughput frontier*.

Visualizing the throughput frontier is straightforward — it can be represented by mapping the hybrid throughputs to 2D space. Figure 1a shows an example of a throughput frontier created by randomly sampling a large number of different workload mixes ((τ,α) pairs) and computing the corresponding hybrid throughputs. The x-axis represents the T throughput measured in completed successful transactions per second (tps). The y-axis represents the analytical throughput measured in completed queries per second (qps). We denote the maximum transactional and analytical throughput as X^T and X^A, respectively. The throughput frontier is always bounded by X^T in the x-axis and by X^A in the y-axis.

This sampling approach to create the throughput frontier can be prohibitively time-consuming. A more systematic way of computing the throughput frontier is illustrated in Figure 1b, called the saturation method. Instead of randomly sampling different workload mixes, wefi x either the T or A clients while varying the number of the other type of clients until the performance stops improving.



(a) PostgreSQL-SR for SF100. **(b)** PostgreSQL-SR for SF100. **(c)** TiDB for SF10. **(d)** System-X for SF1.

Figure 2: Examples of our performance-centric definition: (a) Grid graph and (b, c, d) Throughput frontier.

The vertical and horizontal lines shown in thefi gure correspond to series of measurements where the number of T (or A) clients arefi xed and the number of A (or T) clients is varied. We call them *fixed-T* and *fixed-A* lines respectively. We call the graph formed from thefi xed-T andfi xed-A lines the *grid graph (g)*.

Figure 2a shows a real example of a grid graph created for PostgreSQL streaming replication (PostgreSQL-SR) with a 100 GB dataset; more details of the workload and experiment will be presented in Sections 5 and 6. The real grid graphs do not include pure vertical or horizontal lines. As it shows in Figure 2a, the real fixed-T andfi xed-A lines are sloped and the distances between the individual lines vary. The shape of thefi xed-T andfi xed-A lines can explain the way the T and A components of a workload affect each other when they run concurrently. We provide more details in the interpretation of thefi xed-T andfi xed-A lines in Section 3.2.1. Moreover, in Section 3.3, we will discuss the way Figure 2a was created by introducing an efficient algorithm.

3.2 Interpretation of the throughput frontier

In this section, we discuss the information that can be extracted from the throughput frontier and how this information can be used to interpret the performance of an HTAP system. In general, the throughput frontier quantifies the absolute T- and A-throughput, and their relationship. It is useful for diagnosing performance issues.

To fully understand the performance of an HTAP system, we must consider both the *magnitude* and the *shape* of its throughput frontier. The magnitude of the throughput frontier (i.e., the distance between each point on the frontier and the origin) represents the absolute performance of the system across the entire HTAP workload spectrum. The throughput frontier magnitude is most useful when comparing multiple HTAP systems. If the throughput frontier region for some system A completely envelops that of another system B, we can say that system A offers higher HTAP performance than system B on the given workload. In contrast, it is also possible for neither throughput frontier region to fully contain the other. In this case, we recommend a deeper analysis, which takes into account additional factors such as the expected workload mix, to determine which system is more desirable. The remainder of this section is dedicated to analysis of throughput frontier shape.

To enhance this discussion, we will use examples of throughput frontiers derived from experiments on real systems. Figure 2b, Figure 2c, and Figure 2d show the throughput frontiers of PostgreSQL-SR, TiDB, and a commercial database which we anonymize as System-X. PostgreSQL-SR and System-X use the serializable isolation level while TiDB guarantees snapshot isolated reads. A scaling factor of 100 for the HATtrick benchmark was used for PostgreSQL-SR, 10 for TiDB and 1 for System-X. The total raw data size is roughly 80 GB for PostgreSQL-SR, 10 GB for TiDB, and 1 GB for System-X, in all cases the datafi ts in main memory. More configuration details will be presented in Section 6.

We now introduce two annotations to the throughput graph to better understand the shape of the throughput frontier: the *proportional line* and the *bounding box*. The proportional line (p_f), illustrated by the blue dashed line in Figure 2b and subsequentfi gures, is the line drawn from the two extreme points of the throughput frontier. It represents a relationship of *linear dependence* between T- and A-throughput. The bounding box (b_f), illustrated by the red dashed rectangle in Figure 2b and subsequentfi gures, is the rectangle formed by the extreme points of the throughput frontier (i.e., $0 \leq x \leq X^T$ and $0 \leq y \leq X^A$). The bounding box represents *independence* between T- and A-throughput.

In subsequent paragraphs, we explain how the proportional line and the bounding box aid in the analysis of the throughput frontier. We consider three general throughput frontier patterns: (i) a throughput frontier that is *close to* the proportional line, (ii) a throughput frontier that is *well above* the proportional line and close to the bounding box, and (iii) a throughput frontier that is *well below* the proportional line and close to the axes. Note that a real system may exhibit a throughput frontier with any combination of patterns. Here, we separately consider each pattern only to build intuition about the throughput frontier.

Close to the proportional line. As described earlier, the proportional line represents a linear relationship between T- and A-throughput. The proportional line is named as such to emphasize the tradeoff between T- and A-throughput: in an HTAP system whose throughput frontier remains close to the proportional line, any increase in T-throughput is accompanied by a *proportional* decrease in A-throughput, and vice versa. HTAP systems that exhibit this behavior are attractive for their predictable performance. An example of a system and workload configuration that produces a frontier with this pattern is TiDB with SF10, as shown in Figure 2c.

Above the proportional line, close to the bounding box. As described earlier, the bounding box represents independence between T- and A-throughput. In an HTAP system whose throughput frontier is well above the proportional line and close to the bounding box, it may be possible to increase T-throughput with minimal

impact on A-throughput, and vice versa. HTAP systems that exhibit this behavior are attractive for their performance isolation. An example of a system and workload configuration that produces a frontier with this pattern is PostgreSQL-SR with SF100, as shown in Figure 2b. Note that, by definition, the throughput frontier of every HTAP system will always be within the bounding box.

Below the proportional line, close to the axes. Qualitatively, the degree to which a throughput frontier is below the proportional line and close to the axes represents the amount of negative interference between the T- and A-portions of the workload. A throughput frontier that is well below the proportional line is an indicator of poor HTAP performance and may indicate contention for resources in the system. Identification of this pattern may be useful in diagnosing performance issues. An example of a system and workload configuration that produces a frontier with this pattern is System-X with SF1, as shown in Figure 2d. Importantly, the size of the database in this configuration is comparatively small, which results in increased contention for data items. Wefi nd that HTAP systems generally exhibit throughput frontiers below the proportional line for small database sizes, more discussion in Section 6.

3.2.1 Grid graph. In addition to the throughput frontier, the grid graph provides complementary information regarding *workload preference*, through the slope of thefi xed-T andfi xed-A lines. Ideally, if there is no workload interference, the grid would be comprised of pure vertical and horizontal lines. This is rarely the case in real systems, the lines tend to be slanted due to the interference between the T and A workloads. The closer afi xed-T orfi xed-A line is to be perpendicular to the axes the less the corresponding workload is affected by the increase of the other workload. Figure 2a, shows the grid graph of PostgreSQL-SR which corresponds to the throughput frontier of Figure 2b. Thefi xed-T lines of thefi gure are closer to vertical, are clearly placed and tend to have the same length which reaches the X^A. Thefi xed-A lines are not smooth since they have fluctuations in the absolute numbers of the T-throughput but they tend to have the same length which reaches the X^T. This means that the interference of the T and A workloads is minimized in PostgreSQL-SR in this specific configuration and that PostgreSQL-SR is not favoring a workload over the other.

We also get *workload preference* information from the throughput frontier but the *grid graph* provides more resolution at operation areas below the frontier that might be of interest in practice.

3.3 Calculation of throughput frontier

In Section 3.1, we introduced the saturation method for calculating the throughput frontier (Figure 1b). Here, we describe in detail how it works including the creation of thefi xed-T andfi xed-A lines.

First, wefi nd the number of transactional clients (τ_{max}) that maximize the transactional throughput X^T. Tofi nd (τ_{max}), the HTAP system executes the transactional workload with an increasing number of clients, until the transactional throughput does not further increase. The algorithm repeats the same steps tofi nd the number of analytical clients (α_{max}) that maximize the analytical throughput X^A. Note that for any other different workload mix, the DBMS cannot achieve a transactional or analytical throughput higher than X^T or X^A, respectively.

The next step is to collect the data points that create thefi xed-T andfi xed-A lines. Each line requires a series of measurements, in which the number of T (or A) clients isfi xed and the number of A (or T) clients is varied. In our evaluation we create sixfi xed-T and six fixed-A lines, by equally diving the ranges $[0, \tau_{max}]$ and $[0, \alpha_{max}]$. For each line, we collect six points. We found that this configuration provides a good coverage of the space, but the number of points per line collected as well as the spacing of the lines can be tuned to provide better coverage. After all data is collected, we calculate the throughput frontier. The throughput frontier is made up from the highest point of eachfi xed-T andfi xed-A lines.

4 FRESHNESS OF HTAP SYSTEMS

In addition to the performance-centric definition, we need to highlight the importance of an HTAP system to provide fresh analytics. In this section, we introduce the concept of *freshness score* which is used to describe the recency of the data read by an analytical query. We also describe our method that can be used to measure the *freshness scores* of queries in real database systems.

4.1 Theoretical definition of freshness

We consider again a hypothetical HTAP system that serves a mix of T- and A-clients and each of them issues a constant stream of requests. In this definition, we assume that both the clients and the HTAP database have access to the same global clock. A transaction is considered committed when the updates of the transaction are applied to the database and are visible to the other transactions. Each analytical query starts andfi nishes at a particular time based on the global clock and reads a specific snapshot of the operational data. An up-to-date version of the operational data includes all the updates made by transactions that committed before the start of the analytical query. In contrast, a stale version misses some of such updates. We say an HTAP system provides fresh analytics if every analytical query is executed on an up-to-date version of the operational data. Else, it provides stale analytics.

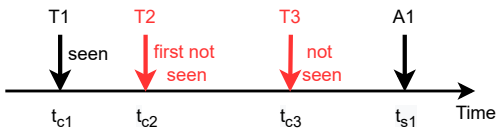


Figure 3: Illustration of freshness for analytical queries.

We define *freshness score* of an analytical query A_q as a quantitative measure $f_{A_q} = max(0, t^s_{A_q} - t^{fns}_{A_q})$. $t^{fns}_{A_q}$ is the commit time of the **f**irst transaction **n**ot **s**een by A_q and $t^s_{A_q}$ is the start time of the A_q. Both measures are based on the global clock. Given the definition, the smaller the measure is, the fresher the system will be. The freshness score of A_q is zero when the query can see the updates from transactions committed before the start of the query, which means the snapshot is up-to-date. When the snapshot is outdated, to calculate the freshness score we need tofi nd the time after which the snapshot became stale. This time is equal to the commit time of thefi rst transaction whose updates are not present in the version of

the data in which A_q runs. Then, the freshness score of A_q is equal to the difference between the start time of the query and thefirst unseen transaction measured in time units, e.g., seconds. Figure 3 shows an example of transactions T1, T2, T3 and an analytical query A1. Each t_{ci} corresponds to the commit time of the transaction i and the t_{s1} corresponds to the start time of the analytical query A1. We assume A1 sees all the changes made by T1 but does not see changes by T2 or T3. Therefore, T2 is thefi rst-not-seen transaction and the freshness score of A1 is $f_{A_q} = t_{s1} - t_{c2}$.

Since the A-clients issue multiple requests, each A_q will have a different freshness score. Thus, we define the *freshness score* of an HTAP system as the aggregation of the freshness scores of all analytical queries, denoted as f_{agg}. agg can be any aggregation function such as the average or 95% percentile. Freshness $f_{avg} = 0$ means that the HTAP system can always provide the most recent version of the operational data to all the analytical queries. Freshness $f_{avg} = p$ seconds means that on average the snapshot used by the analytical queries is out-dated by p seconds.

4.2 Measuring freshness score

The theoretical definition of freshness defined in the previous section can be challenging to measure in a practical system. In particular, we identify the following two challenges:

Challenge 1: No global clock. The theoretical definition of freshness score requires a global clock that is accessible from both clients and the database. A practical system, however, does not have an accurately synchronized clock across different nodes, making it difficult to measure commit time or query start time.

Challenge 2: Hard to identifyfi rst not seen transaction. The definition of freshness score requires identifying thefi rst transaction that is not seen by each analytical query. This task is particularly difficult since by definition, the analytical query cannot identify such a transaction. Extra bookkeeping information needs to be kept to identify a not seen transaction.

We introduce the following new algorithm to approximate the theoretical freshness score of a query and resolve the two challenges above. The algorithm has minimal impact to the workload in terms of modifications, and can be applied to general HTAP benchmarks.

To resolve thefi rst challenge, we decide to conduct all time measurement on the client side. In particular, the commit time of each transaction is the time when the transaction result is returned to a client. The start time of an analytical query is the time when the query is sent to the database. This solution avoids clock synchronization across database nodes, and the freshness score is consistent with what the client observes.

To solve the second challenge, we need tofi rst ensure that a client knows which transactions each analytical query should observe, and second be able to tell which transactions the analytical query actually observed based on the returned result. We introduce a set of lightweight tables $FRESHNESS_j$, where $j \in [1, \tau]$ and τ is the number of transactional clients. For each transactional client j we create one such table that acts as a synchronization point. We also update the transactions and analytical queries in the workload such that they update and read the corresponding table.

Each $FRESHNESS_j$ table contains only one integerfi eld, which is the ID of the last transaction from transactional client j. Each transaction will execute extra logic to update the $FRESHNESS_j$ table with its ID. Note that a transactional client submits transactions to the database sequentially with increasing IDs. Therefore, at most one transaction will be updating each $FRESHNESS_j$ table at any given time and the ID in the table will monotonically increase. We deliberately design the $FRESHNESS_j$ tables to be separate (one for each client) instead of storing multiple rows in a single table in order to reduce contention from having different clients updating their IDs concurrently. Thus, the transactional latency is not affected by the table locking protocol of each database.

To identify which transactions are observed by an analytical query, we modify each query to read all $FRESHNESS_j$ tables and return the contents to the client. Specifically, we union the $FRESHNESS_j$ tables and cross-join the result with the original query. If a query is executed against a consistent snapshot, the returned IDs define the transactions observed by the query — transactions with larger IDs are not observed by the query. This way, we successfully identify thefi rst-not-seen transaction and can calculate the freshness score.

Note that the algorithm described above works well when analytical queries are serializable or snapshot isolated, where reads of data tables and freshness tables are consistent within a query. If the queries are executed with lower isolation levels, one way to measure freshness is to embed the $FRESHNESS_j$ information into each tuple at the cost of higher overhead. All the systems we measure run with at least snapshot isolation and therefore we maintain $FRESHNESS_j$ as separate tables.

5 DESIGN OF HATTRICK BENCHMARK

We will now move to the design of our hybrid benchmark called *HATtrick* which is an open source project [1]. HATtrick complements the throughput frontier, incorporates our freshness measurement method, and can be used to effectively evaluate HTAP systems.

The HATtrick benchmark contains an analytical component and a transactional component. The analytical component is based on the Star-Schema Benchmark (SSB) [26]. We extend the SSB schema to support a new transactional workload which is an adapted version of the TPC-C [3] benchmark. This section discusses the schema, the workload, and the implementation details of HATtrick.

5.1 The schema and data

Figure 4 shows the schema of the HATtrick benchmark, which keeps all the SSB entities and relationships almost unmodified. We update the CUSTOMER, SUPPLIER, and PART relations by adding one new attribute to each of them. Also, we introduce a new relation called HISTORY and a series of relations called $FRESHNESS_j$, where $j \in [1, \tau]$ and τ the number of transactional clients. The purpose of adding the new attributes and the HISTORY relation is to support the transactional workload component of HATtrick; the $FRESHNESS_j$ relations are used in the freshness measurement process as described in Section 4.2. Each $FRESHNESS_j$ table contains only one integerfi eld, the TXNNUM.

Specifically, we add the attribute PAYMENTCNT in the CUSTOMER relation which is an integer that keeps track of the total number of payments each customer makes. Also, we add the attribute YTD in the SUPPLIER relation which is a decimal that

[1] https://github.com/UWHustle/HATtrick

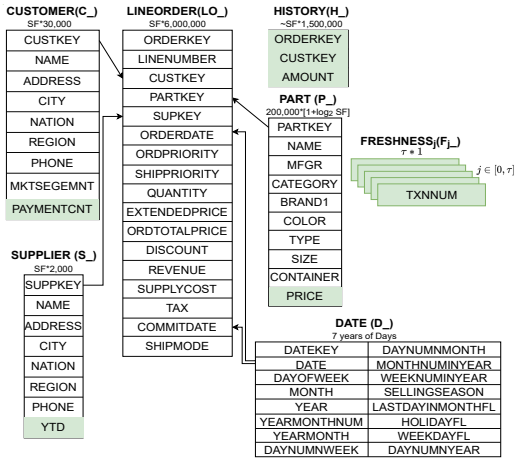


Figure 4: The schema of HATtrick benchmark based on modified SSB. New attributes in HATtrick are in shade.

accumulates the year to date profits of each specific supplier. Both these attributes are going to be used and updated in transactions which are similar to the payment transaction in TPC-C benchmark.

The HISTORY relation consists of three attributes, the ORDERKEY from the LINEORDER relation, the CUSTKEY from the CUSTOMER relation, and the AMOUNT which is a new decimal attribute. An insertion in the HISTORY relation simulates the process of keeping historic information for a customer payment.

The last change is made in the PART relation where we added the PRICE attribute which is a decimal that stores the cost of each part. The PRICE attribute is used in every transaction that inserts new orders in the LINEORDER relation when the EXTENDEDPRICE and ORDERTOTALPRICE attributes are computed. We describe in Section 5.2 how exactly these additions are used in each transaction.

HATtrick benchmark follows the scaling of the SSB benchmark for the initial population of the database, Figure 4 shows more details. After the initial population, the sizes of the CUSTOMER, SUPPLIER, PART, and DATE relations remain unaffected by the T workload. However, the transactions of HATtrick change the sizes of the LINEORDER and HISTORY relations by adding new tuples. The initial size of the HISTORY relation equals the number of the unique ORDERKEYs in the LINEORDER relation, that number is approximately the 25% of the size of LINEORDER relation. The size of each $FRESHNESS_j$ relation isfi xed and equal to one.

5.2 Workload

There are two components in the HATtrick benchmark, the analytical and the transactional.

5.2.1 Transactions. The HATtrick benchmark defines three transactions modeled after the TPC-C benchmark. Specifically:
New order: This transaction enters a complete order with multiple lineorders through a single database transaction. The new order is inserted to the LINEORDER relation. Specifically, given a random customer name C_NAME, part key P_PARTKEY, supplier name S_NAME, and day of order D_DATE, the new order transaction reads the CUSTOMER, PART, SUPPLIER, and DATE relations to retrieve data. These data are used to create the new entries of the LINEORDER relation. For example, based on the P_PARTKEY, a P_PRICE is retrieved which is used to compute the attributes EXTENDEDPRICE and ORDERTOTALPRICE for that specific lineorder. It is worth mentioning that, the dates are sampled from the fixed range of the DATE relation which is seven years of days from 1992 to 1998. Therefore, the new line-orders that are added through the new order transaction do not insert new dates but they keep sampling uniformly from thefi xed range.
Payment: An update transaction which simulates a customer's payment for an order that they already made. The transaction updates the customer's total number of paid orders and the year to date balance of the order's supplier which correspond to the C_PAYMENTCNT and S_YTD attributes respectively. The transaction commits after inserting the payment information to the HISTORY relation. The customer is selected by customer name C_NAME 60% of the time and by the customer key C_CUSTKEY the rest of the time.
Count orders: A read-only transaction which reports the total number of orders for a given customer. The customer is selected by C_NAME, so seeking on the secondary index of the CUSTOMER relation is required. The total number of customer's orders is retrieved from the LINEORDER relation.

Each transaction generated by the T-client j, additionally to its original workload updates the $FRESHNESS_j$ relation with the transaction's ID.

5.2.2 Analytical queries. The analytical component of the HATtrick benchmark includes all the 13 queries of the SSB benchmark modified to also return the data from the $FRESHNESS_j$ relations. During the measurement period, the transactions add new orders to the LINEORDER relation and thus, the analytical queries process more rows as the time passes. The new entries added through the New Order transaction follow the same specifications as defined in the SSB benchmark.

5.3 Benchmark procedure

During operation an HTAP system evaluates transactional requests and analytical requests simultaneously. Each client issues a transaction or a analytical query based on their type and waits for the result before issuing the next. The number of clients is not restricted but the ratio of T to A clients (T:A) is a benchmark parameter.

The T clients issue transactions with the following distribution: 48% New Order, 48% Payment and 4% Count Orders. Each client is independent of other clients. The A clients' queries are organized in batches. An A batch contains all the 13 queries ordered randomly. Once all the queries in the batch havefi nished execution, the A client continues with a new batch of the 13 queries and a new permutation of them. The A queries do not delay the transactions, which could happen if a client runs both types of queries. With this design, the tested database systems are free to delay the transactions or the analytical queries in order to improve performance.

6 EXPERIMENTAL EVALUATION

In the evaluation, we experiment with different databases and configurations. Specifically we study how performance and freshness scores change for different database sizes, isolation levels, physical

schemas, replication modes, and deployments (single node and distributed). We use PostgreSQL [28, 39, 40], PostgreSQL Streaming Replication [27] (PostgreSQL-SR), an anonymized System-X, and TiDB [15] for this part of the evaluation. Before we present the experiments, we describe the experimental configuration and the setup of each database system that we use.

6.1 Experimental configuration

System configuration. The single-node and multiple-node experiments are performed on the same type of servers. Each server has a 2.35Ghz AMD EPYC^{TM} 7452 processor with 32 physical cores, a 512GB RAM, and an SSD disk and runs the Ubuntu 18.04 LTS.

Benchmark configuration. We experiment with three scale factors of the HATtrick benchmark, SF1, SF10, and SF100, that correspond to raw data of sizes 570MB, 5.7GB, and 59GB respectively. For all scaling factors and database systems that we tested, the data alwaysfi ts in memory. The clients that submit the transactional and the analytical requests run on the same machine in which the tested database system is installed. For multi-node setup, the clients of the benchmark run in one of the nodes (e.g., in PostgreSQL-SR, the clients run on the same machine with the primary node).

The duration of each benchmark run consists of a warm-up period and the measurement period. Each scaling factor has a different warmup and measurement period duration. For example, for SF100 the warm-up duration is 5min and the real measurement phase is 10min. For SF10 the warm-up is 3min and the measurement phase is 6min and for SF1 2min warm-up and 4min measurement phase. The duration of each period for each scaling factor was selected after conducting a small experiment, where we discover the appropriate time periods for each phase so the performance is stable. The duration of the warmup and measurement periods remain the same across systems when experimenting with the same scaling factors. Before each benchmark run we reset the data to their initial state.

For each workload configuration (T:A client ratio) we repeat the execution of the benchmark three times and report the average results. For each workload configuration the benchmark reports the T throughput in successful transactions per second (tps) and A throughput infi nished queries per second (qps). We also compute freshness score for each T:A client ratio. HATtrick benchmark extracts also the average response time of each transaction type and analytical query.

Evaluated systems configuration. The databases we use are PostgreSQL 14, System-X, and TiDB 5.2.0. Due to legal restrictions, we do not disclose the original name of System-X. In PostgreSQL and PostgreSQL-SR, we created all possible B+ tree indexes on the attributes used in the predicates of the transactional and analytical requests. We used this configuration to accelerate both workloads in PostgreSQL for all the experiments except for the one in which different physical schemas are tested. In System-X and TiDB, we created all needed B+ tree indexes for accelerating the transactional requests. Both System-X and TiDB provide an additional column based representation of the data to speed up the analytical requests. Stored procedures were used to execute the transactional requests and prepared statements to execute the analytical requests in PostgreSQL, PostgreSQL-SR, and System-X. Prepared statements were used to execute both the transactional and analytical requests in TiDB since stored procedures are not yet available. Finally, for all databases we disabled the option of intra-query parallelism since it leads to over-utilization of the resources when multiple analytical requests are executed in the database.

Reported results. For each experimental configuration, we report three plots that correspond to thefi xed-T lines, thefi xed-A lines, and the throughput frontier, respectively. We generate thefi xed-T andfi xed-A plots as described in Section 3.3. We then compute the throughput frontier from thefi xed-T andfi xed-A data, also as described in Section 3.3. Eachfi gure in this section was generated with this method. In addition to the throughput frontier we also compute the freshness scores as described in Section 4.2. We report the 99th-percentile of the freshness scores for the T:A client ratio points 20:80 (f_2), 50:50 (f_5) and 80:20 (f_8) measured in seconds.

6.2 PostgreSQL

In this section we run the HATtrick benchmark in PostgreSQL 14 and show results for different scale factors, isolation levels, and physical schemas. Our results show that there is a negative interference between the T and A workloads in all the scale factors, isolation levels, and schemas. This leads to a throughput frontier that is either below or close to the proportional line. Finally, PostgreSQL is able to provide a zero freshness score in all the experiments, which is expected based on its architecture.

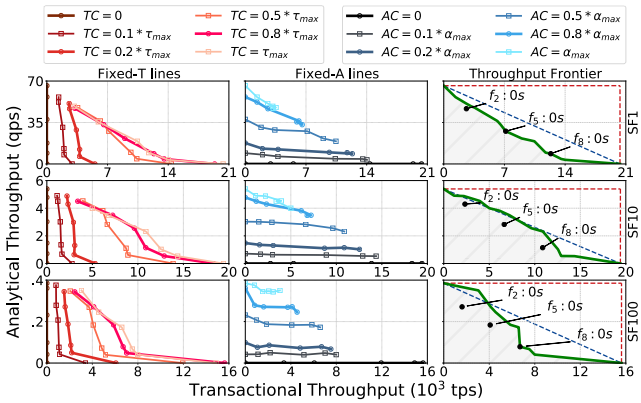


Figure 5: PostgreSQL for different scaling factors.

System design. PostgreSQL is a relational database management system (RDBMS) designed primarily for transactional processing. However, like many other traditional databases, PostgreSQL can also serve hybrid workloads. PostgreSQL uses multiversion concurrency control (MVCC) in which readers never block writers, and vice versa. In this set of experiments, we use the serializable isolation level.

Varying scaling factors. Figures 5 shows the performance results of HATtrick benchmark for PostgreSQL in three different scaling factors. Thefi xed-T lines and thefi xed-A lines for SF1 have a non smooth behavior. As the number offi xed T/A clients is increased, the lines become more slanted. This shows that the increase of the T(A) clients across thefi xed-A(fixed-T) lines affects negatively the A(T) throughput. In general, the behavior of thefi xed-T and fixed-A lines shows that as the number of the T and A clients

increases, the two workloads are competing for compute resources and data. We observe this behavior — throughput frontier below the proportional line for SF1 — for all the tested databases of Section 6. The small size of the database contributes more to this behavior [30, 34, 42] since many transactions update the same rows which due to locking leads to increased waiting times. The throughput frontier for SF1 is always below the proportional line, suggesting negative interference between the two workloads.

Moving to SF10, thefi xed-T andfi xed-A lines continue to have a slanted behavior. However, the frontier is now moving closer to the proportional line. This means that an increase in the T throughput is accompanied by a proportional decrease in A throughput and vice versa. Thus, the resource sharing between the two workloads is more efficient than in SF1. Compared to SF1, we see a big drop in the maximum A throughput due to the increase of the database size which leads to bigger answer sizes. In terms of maximum T output there is also a slight reduction compared to SF1.

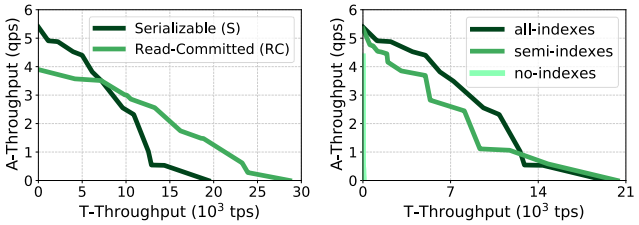
In SF100, thefi xed-A lines are not significantly affected by the increase of the T clients. In contrast, thefi xed-T lines are the ones which are extremely affected by the increase of the A clients. Thus, thefi xed-A lines tent to be parallel and to all have the same length, while thefi xed-T lines have a slanted behavior. As a result, the throughput frontier of SF100 is for the half part above or close to the proportional line and for the rest part below the proportional line. This indicates bad performance scaling and shows that the database system is not able to efficiently serve the two workloads in parallel because the T throughput is extremely affected by the increase of the A clients. Again, there is a drop in the maximum A throughput compared to SF10 which is related to the increase of the database size. Interestingly, we observe a significant decrease of the maximum T throughput compared to SF10. This is related to the big number of B+ tree indexes that we use to accelerate both the T and A parts of the workload. However, as the size of the database increases, the size of the indexes increases too. Therefore, more time is needed to traverse and update the indexes when the transactions are executed, leading to a degradation of the T throughput.

The measured freshness score for all the ratio points and scale factors is equal to zero. This is expected since PostgreSQL maintains one copy of the data and the updates of the transactions are made immediately available to the data snapshot used by the A queries.

The advantage of PostgreSQL is that analytical requests can run concurrently with the transactional requests by using snapshot isolation. However, the two workloads still need to compete for resources, data structures, and data items, and this becomes worse when the number of the T and A clients are both high.

Varying isolation levels. We now use PostgreSQL and experiment with different isolation levels. We show how the throughput frontier captures the behavior differences between isolation levels.

Figure 6a shows the throughput frontiers of PostgreSQL in serializable and read committed isolation levels for SF10. The read committed isolation level achieves higher T and A throughput in almost all the parts of the throughput frontier. The throughput frontier of the serializable isolation level achieves a better maximum A throughput. This is because the PostgreSQL query optimizer chooses different plans for the analytical queries in different isolation levels. However, in all the other cases the serializable throughput frontier is always below the read committed throughput frontier. This is an expected result and this experiment demonstrates how throughput frontier reveals the behavior of a system in different isolation levels. Another important observation is the position of the two throughput frontiers relative to their proportional line — both throughput frontiers are close to their proportional lines.



(a) Different isolation levels. **(b)** Different physical schemas.

Figure 6: Within system experiments for PostgreSQL.

Varying physical schemas. We now experiment with different physical schemas in PostgreSQL. The throughput graphs helps us understand the advantages and disadvantages of each physical schema. The results coincide with what we expected to see.

Figure 6b shows the throughput graphs when the physical schema of the database changes; the experiment is performed with serializable isolation level and SF10. The three different physical schemas are the (1) no indexes, (2) with B+ tree indexes that accelerate only the T workload (semi indexes) and (3) with all possible B+ tree indexes that can accelerate the T and the A workload.

In terms of performance scaling, the physical schema with all the possible B+ tree indexes achieves the best results since the throughput frontier is almost always above the throughput frontiers of the other physical schemas. Next in ranking is the physical schema with the semi B+ tree indexes and the worst is the no indexes physical schema. After conducting an analysis we conclude that the different shapes in the throughput frontiers of the three physical schemas are due to the different query plans the optimizer creates for the analytical queries based on the available indexes.

In terms of maximum T throughput, the semi indexes schema achieves better performance compared to the all indexes schema. More indexes can affect the T throughput since they need to be updated in every change that transactions make. However, for the rest workload mixes the all indexes and the semi indexes schemas achieve similar T throughputs.

The use of indexes seem to help not only the T workload but also the A queries. PostgreSQL achieves the best results in both workloads when it uses the all indexes physical schema and this is demonstrated by the throughput frontier results.

In both experiments above (i.e., varying isolation levels and varying physical schemas), we show how the throughput frontier can be used for choosing among different database configurations. Our method combines all the needed information in onefi gure for multiple configurations, thus the users can understand the system's behavior easily and draw conclusions faster.

6.3 PostgreSQL streaming replication

In this section we use PostgreSQL 14 with streaming replication (PostgreSQL-SR) and we run the HATtrick benchmark for different

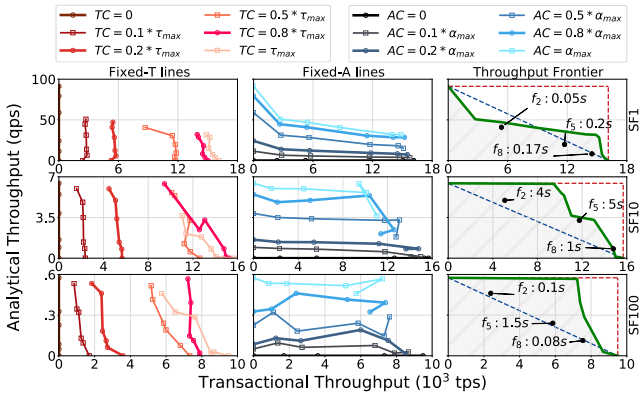


Figure 7: PostgreSQL-SR for different scaling factors.

scale factors and replication modes. The results show that as the scale factor increases the throughput frontier moves above the proportional line, indicating good performance scaling and shows that the database system is able to concurrently serve the two workloads efficiently. However, in all the scale factors we experienced stale queries. Also, the experiments with different replication modes show a trade-off between performance and the freshness scores.

System design. Streaming replication is the most common PostgreSQL replication strategy in which a primary node replicates data to the standby server(s). It is based on streaming WAL records to the standby server(s) as they are generated without waiting for the WALfi le to befi lled. Thus, it allows the standby server(s) to stay more up-to-date than with thefi le-based log shipping. The primary node is usually used for transactional workloads while the standby node(s) is read only.

By default PostgreSQL streaming replication is asynchronous, which means that there is a small delay between committing a transaction in the primary and the changes becoming visible in the standby node(s). However, a user can set up different replication modes. One option is the strict synchronous replication in which a transaction in the primary commits only after the updates are replayed in the standby node(s). This mode can be chosen by setting the synchronous_commit parameter of PostgreSQL-SR to remote_apply. We call this mode RA. In this mode, one can use PostgreSQL-SR to execute HTAP workloads and provide analytics with freshness score equal to zero.

In ourfi rst part of the experiments we choose to relax the replication mode and set the synchronous_commit parameter to ON. We call this mode ON. In ON mode, a transaction in the primary node will commit only after the standby server(s) confirms that the transaction record was safely written to the disk of the standby server(s). The difference between the RA and ON mode is that in the ON mode the transmission of the updates happens synchronously but the actual replay of the updates is asynchronous. In RA mode both steps happen synchronously by the commit time of the transaction. Since the transaction updates in ON mode are replayed asynchronously, we expect to see some stale queries.

Varying scaling factors. In this experiment, we set up PostgreSQL-SR in two identical nodes — one is the primary and is responsible for the transactional workload and the other is the standby responsible for the analytical workload. We choose replication mode ON.

Figure 7 shows the results for different scale factors. Thefi xed-T andfi xed-A lines for all the scale factors are less slanted compared to the PostgreSQL experiments of Section 6.2. This behavior is more clear in the cases of SF10 and SF100 where the lines tend to be parallel and have the same length. This means that the $T(A)$ workload tend to be less affected by the increase of the $A(T)$ clients. Thus, as the scale factor increases the throughput frontier moves above the proportional line and in SF100 is close to the bounding box. These results show that PostgreSQL-SR is good at isolating the performance of T and A workloads and can serve the two workloads efficiently. Interestingly, the results are representative of the system's architecture. The primary and standby nodes have isolated resources and thus the interference of the T and A workloads is expected to be limited compared to systems with shared resources.

As the scale factor increases we see a decrease in the maximum A throughput. In terms of the maximum T throughput there is a significant decrease in SF100 compared to SF1 and SF10. This is related to the increased size of the indexes as discussed in Section 6.2.

In Figure 7 we report the freshness scores for all the scaling factors in the three T:A client ratios. However, we cannot compare the absolute freshness score of a specific T:A ratio across the different scale factors since they correspond to different number of clients.

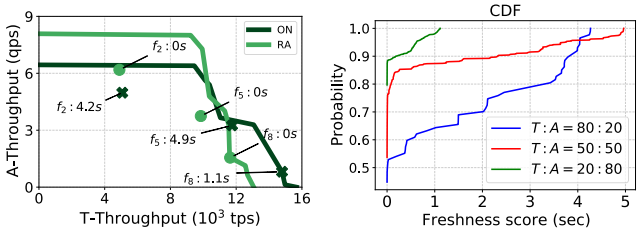
In Figure 8b we show the CDFs of the freshness scores for the three ratios in SF10. For the client ratio 20:80, almost 90% of the executed queries return freshness score close to zero and the maximum freshness score seen is 1.1sec. Moving to the client ratio 50:50, the results show that 75% of the executed queries return freshness score close to zero and the maximum freshness score seen is 4.9sec. Finally, the client ratio 80:20 execution reports almost 55% of the queries with freshness score close to zero and the maximum value seen is 4.2 sec. These results indicate that the freshness scores are significantly affected by the number of the T clients. For example, the ratio 80:20 has the lowest percentage of fresh queries ($\sim$ 55%). This is reasonable since more transactional clients are performing more updates in the primary node which need to be send and applied to the standby replica. As a result, the standby node cannot keep up with the high rate of updates and thus, the analytical queries are executed in more outdated snapshots.

Varying replication mode. Next we experiment with different replication modes in PostgreSQL-SR and SF10. Results show that the performance of a system can be affected by the provided freshness.

Figure 8a shows the throughput frontiers in SF10 for two different replication modes, ON and RA. Thefi gure includes also the freshness scores for the three client ratios in each mode. In RA mode every transaction in the primary server has to wait for the updates to be applied in the standby node before committing. Thus, the standby node remains always up-to-date and the freshness scores are equal to zero for every query. On the contrary, in the ON mode the replay of the data in the standby server happens asynchronously and thus we see stale queries. Both the throughput frontiers of the ON and RA modes are above their proportional lines which means that the system in both modes can efficiently isolate the performance of T and A workloads. In thefi rst half part of the frontiers the RA is above the ON frontier and for the rest half the ON frontier is above the RA. This means that in the RA mode more analytical queries are executed and in the ON mode more transactions. This is because the RA mode affects the latency of

the transactions in the primary node and thus the T throughput is lower. The fact that less transactions are executed in the RA mode, leads also to a small increase in the A throughput.

In this experiment we see a trade-off between freshness and performance. To achieve fresh analytical queries, T performance is sacrificed. This trade-off can be easily understood by using our throughput frontier graphs and freshness measurements. Using the proposed metrics users can choose the appropriate configuration based on their preferences and application requirements.



(a) Different replication modes. **(b)** CDFs for mode ON.

Figure 8: Freshness results for PostgreSQL-SR.

6.4 System-X

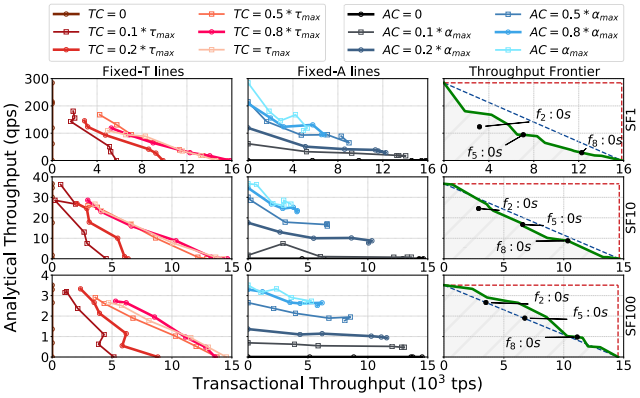


Figure 9: System-X for different scaling factors.

In this section we use System-X to run the HATtrick benchmark for different scale factors. The results show that System-X can guarantee freshness and as the size of the database increases, it becomes more efficient in handling the two workloads concurrently.

System design. System-X is a memory optimized engine designed to accelerate transactions. The experiments use the serializable isolation level which is achieved by optimistic MVCC without locking. The internal data structures are all latch-free and the threads are executed without stalling or waiting. System-X provides clustered column store indexes which can be also stored in memory and used to accelerate the A workload. When System-X is used for hybrid workloads, it can be configured to maintain two copies of the data in memory with each copy having a different data representation. Therefore, transactions can use the row store while analytical queries the column store copy.

Varying scaling factors. Figure 9 shows the results for different scale factors. We identify similar patterns with the PostgreSQL results in Figure 5. Both thefi xed-T andfi xed-A lines are slanted which means that the T and A workloads are affecting negatively each other in all the scale factors. However, in System-X and SF100 thefi xed-T lines are less affected by the increase of the A clients. Thus, the frontier of SF100 is above or close to the proportional line. Also, the column format of the data and the high efficiency of data compression boost the performance of analytics in System-X compared to PostgreSQL. In terms of maximum T throughput, System-X is able to provide an almost stable performance in all scaling factors since the transactional part does not need to "pay" any cost for keeping the analytical data fresh.

Although System-X is lock and latch free and maintains two copies of data, the throughput frontiers of SF1 and SF10 capture competition for resources. It is important to mention that, transactions before committing in System-X need to pass a validation phase in which they validate their reads. If a transaction X is in validation phase and another transaction Y reads the changes X made, then Y becomes dependent on X and it blocks until X commits. When many T clients compete for modifying common data, especially in smaller database sizes (e.g, SF1 and SF10), the blocked transactions that are waiting to commit or abort are more numerous. This affects the T throughput as well as the A throughput since each analytical query must synchronize with transaction updates that have not yet been merged with the column store copy.

It is important to mention that the frontier of System-X is representative of the system's design. System-X maintains two copy of the data to boost the performance of each workload. However, both workloads share the same resources and thus, the shape of the frontier in SF100 is above or close to the proportional line. Compared to the frontier of PostgreSQL for SF100 of Figure 5 the scaling of System-X is better. This is expected since PostgreSQL has only one copy of the data and it is in row format.

For all scale factors, the freshness scores for the three client ratios are equal to zero. This is expected for System-X since based on its design, the latest updates from the operational data are always merged with the analytical data before the execution of a query.

6.5 TiDB

In this section we use TiDB and run HATtrick benchmark for different scale factos and deployment configurations (single node and distributed nodes). Our results show that TiDB can always guarantee fresh analytics. In terms of performance TiDB can serve efficiently the T and A workloads as the size of the database increases.

System design. TiDB is a Raft based HTAP system with a distributed storage layer. The storage layer consists of a row-based store called TiKV and a column-based store called TiFlash. The data stored in TiKV is an ordered key-value map partitioned into many *Regions*. Each Region has multiple replicas and a Raft consensus algorithm is used to keep the replicas consistent within a Region. The replicas of each region form a Raft group which is composed of a leader and followers. Each Raft group has also a learner node which asynchronously receives Raft logs from the leader of the group and transform the row-format tuples to columnar format. More specifically, the learner nodes receive a package of logs from

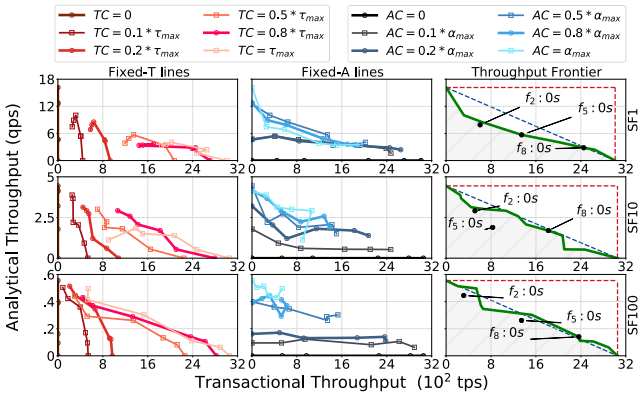


Figure 10: Single node TiDB for different scaling factors.

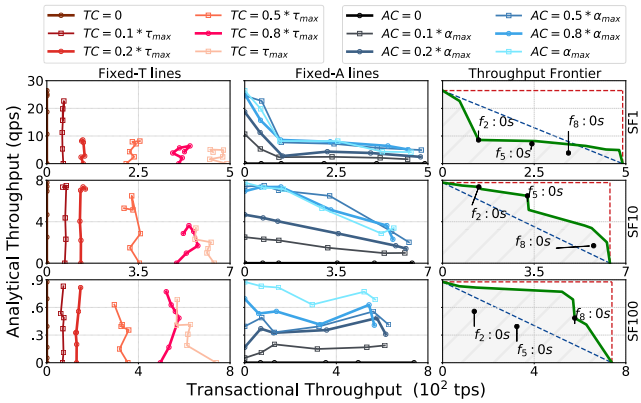


Figure 11: Distributed TiDB for different scaling factors.

the leader which they need to preprocess, decode into row-format tuples, and transform to columnar format. This replication from TiKV to TiFlash makes the fresh data available to the analytical queries and keeps synchronized the two copies.

6.5.1 Single node. Although TiDB is a distributed database, we chose the one server configuration for this experiment. We schedule the transactions to access the TiKV storage and the analytical queries to access the TiFlash storage. We choose for all the experiments the default isolation level of TiDB which is the repeatable read with snapshot isolated reads.

Varying scaling factors. Figure 10 shows the results of TiDB. We identify similar behavior with System-X in thefi xed-T lines,fi xed-A lines and the throughput frontier. In general as the size of the database increases the frontier moves closer to the proportional line. Similar to System-X, TiDB maintains two copies of data in different formats. Although the T and A workloads are executed in different copies, the two workloads share resources. Thus, the frontier for SF100 has a shape above or close to the proportional line. In terms of maximum A performance, we see a drop in the absolute value which is related to the increase of the database size. The maximum T throughput remains almost stable across the different scale factors.

In all the scaling factors the measured freshness scores equal to zero. TiDB is designed to always merge the tail of the log with the analytical data before the execution of an analytical query. Therefore, the latest operational updates are always available to the snapshot of the analytical queries.

6.5.2 Distributed nodes. In this experiment we deploy TiDB in distributed mode. TiKV is deployed in three servers and TiFlash in two servers. TiKV serves the transactional requests and TiFlash the analytical requests. The results show that TiDB in distributed deployment can always provide fresh analytics. Also, it achieves a frontier above the proportional line for SF10 and SF100.

Varying scaling factors. Figure 11 shows the results of distributed TiDB for the three scale factors. Thefi xed-T andfi xed-A lines have similar behavior to PostgreSQL-SR in Figure 7. As the size of the database increases the negative interference of the T and A workloads is minimized and the frontier moves above the proportional line and close to the bounding box.

Compared to the results of Figure 10, TiDB in distributed deployment achieves good performance scaling. The shape of the frontier in the distributed deployment is representative of the system's architecture and shows that the system is close to achieving performance isolation. In terms of maximum T throughput there is a significant decrease compared to the one node TiDB which is caused by the high CPU-overhead of the TCP/IP stack and the limited network bandwidth. However, there is an increase in the maximum A throughput in the distributed deployment which is attributed to the more available resources in the TiFlash component.

6.6 Comparison across systems

In this section we compare all the HTAP systems evaluated above. When comparing different HTAP systems, the process of computing the throughput frontier and freshness scores of each system remains the same. For the comparison we follow a simple rule: If the throughput frontier region of a system A completely envelops that of another system B and system A has lower or same freshness scores compared to B, then system A is better. If not, then we need to dig into more details and to also consider application requirements. Including all the frontiers in onefi gure helps the user to extract conclusions faster. Figure 12 shows the throughput frontiers of PostgreSQL (one node), PostgreSQL-SR (two nodes), System-X (one node), TiDB (one node), and TiDB-Dist (ten nodes) running HATtrick with SF100. Also, we choose to include the freshness scores for the T:A=50:50 client ratio point for each system. Note that systems in thisfi gure may use different number of nodes; we use this example as a point of reference for users that will use HATtrick to compare performance across systems.

From Figure 12, we see that the throughput frontier of System-X envelops the throughput frontiers of all the other systems except for the case of PostgreSQL which has higher value of T_{max} throughput. However, System-X has better A-throughput values and better performance scaling compared to PostgreSQL since its frontier is close to or above the proportional line. We can say that with the workload under test, System-X has the best HTAP performance compared to the other systems.

Between PostgreSQL and PostgreSQL-SR, atfi rst glance, it may not be clear as to which system is better. PostgreSQL-SR has a "higher" frontier and has better A-throughput values, but PostgreSQL has better T-throughput values. Also, PostgreSQL-SR has

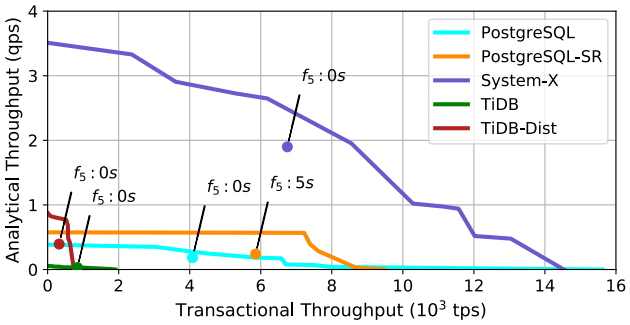


Figure 12: Throughput frontiers of compared systems.

better performance scaling since the frontier is above its proportional line, while PostgreSQL's frontier — mostly below its proportional line — reveals that the T workload is highly affected by the A workload. Furthermore, PostgreSQL-SR cannot always provide fresh analytics while PostgreSQL does. Finally, PostgreSQL-SR uses twice the amount of hardware resources. Deciding between PostgreSQL and PostgreSQL-SR depends on the user's preferences and the application requirements. If the application requires fresh analytics then PostgreSQL is a better choice. However, if the freshness requirements are not so strict then the application can benefit from the better performance scaling of PostgreSQL-SR.

Finally, between TiDB and TiDB-Dist, TIDB-Dist has better performance scaling and A-throughput values. However, TiDB has better T-throughput values. This behavior is expected since TiDB-Dist has distributed transactions. In overall, TiDB-Dist has better HTAP performance compared to TiDB.

6.7 Discussion

To summarize, the HATtrick benchmark can reveal various aspects of an HTAP system and it can also be used to compare different HTAP systems. Specifically, HATtrick can discover information related to absolute T and A throughput, performance scaling in the hybrid workload, the interference of the T and A workloads, and the freshness of the database system. HATtrick combines the above information into a few simple metrics and presents them in a user friendly way, making the process of comparing different HTAP systems easier and more insightful.

We learned the following lessons when conducting this study. First, many current HTAP systems can provide fresh analytics, but this comes with a cost in the T or/and A performance. Second, the results show that the T-throughput is usually severely affected by the number of A-clients; in contrast, the A-throughput is less affected by the number of T-clients. Finally, current HTAP systems cannot achieve complete isolation between the T and A workloads. Future HTAP systems could aim to achieve better isolation between T and A workloads and minimize the impact on the freshness of the analytical query results.

7 RELATED WORK

Recent work on benchmarking HTAP systems includes CH-Benchmark[31], HTAPBench [7] and Swarm64-[29]. Their schema is a combination of the TPC-C and the TPC-H benchmarks. The transactions and the analytical requests remain almost unchanged as in the original TPC-C and TPC-H benchmarks respectively.

CH-Benchmark uses the T and A performance along with the CPU utilization as metrics of the benchmark. The authors use the CH-Benchmark to discover how the freshness of the data, thefl exibility in the transactions features or expressiveness, and the scheduling of the two workloads affect the performance of the HTAP system. They use Hyper [17] and SAP HANA [10, 37] for their evaluation. The results show that fresh analytical queries can result in a degradation of the system's performance. On the contrary, flexibility and scheduling can boost the T and A throughput.

The difference between HTAPBench and Swarm64 compared to CH-Benchmark is that they view one of the workloads as a primary. Usually the analytical workload is viewed as the disturbance of the transactional workload. The users of the HTAPBench and Swarm64 benchmarks specify a target throughput for the primary workload. Then the benchmarks constantly increase the A queries as soon as they do not affect the target throughput.

HTAPBench and Swarm64 propose a method for generating both new data and requests so that the A queries over recently updated data are comparable across runs. One difference between HTAPBench and Swarm64 is the way the distances between timestamps are computed. In HTAPBench the distances in the timestamps are computed without the need for a training run. They use the fact that the data interval in TPC-H isfi xed in all the scale factors. Thus, they use the number of orders in this interval to compute the average time distance between transactions. On the contrary, HTAPBench requires a training run for generating a linear scaling for the timestamps which is then used during the initial data population phase and execution phases.

8 CONCLUSION

In this paper, we introduce a systematic way to evaluate different HTAP systems. We introduce two new metrics, the throughput frontier and the freshness score. The throughput frontier is a 2D graph that captures the overall performance of a database system in the HTAP space. The freshness score quantifies the recency of the data used by the analytical queries. We also propose a method to measure the freshness score of every HTAP system. We validate these metrics by designing a hybrid benchmark called HATtrick and test it in three different HTAP databases. The results show that the throughput frontier is able to show the performance scaling and the interference of the T and A workloads. Moreover, the throughput frontier can discover the design category of an HTAP system and our method for measuring freshness captures the real freshness each HTAP system provides.

9 ACKNOWLEDGMENTS

This work was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program, sponsored by MARCO and DARPA. Additional support was provided by the National Science Foundation (NSF) under grant OAC-1835446 and the Parallel and Concurrent Computer Hardware and Software Research Fund generously provided by an alumnus. Yannis Chronis is supported by a Facebook Fellowship.

REFERENCES

[1] Version 1.14.0. 2011. TPC BENCHMARK™ E.
[2] Vesion 3.0.0. 2011. TPH BENCHMARK™ H.
[3] Revision 5.11. 2009. TPC BENCHMARK™ C.
[4] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research*.
[5] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*. 583–598.
[6] Ronald Barber, Christian Garcia-Arellano, Ronen Grosman, Rene Mueller, Vijayshankar Raman, Richard Sidle, Matt Spilchen, Adam J Storm, Yuanyuan Tian, Pinar Tözün, et al. 2017. Evolving Databases for New-Gen Big Data Applications.. In *CIDR*.
[7] 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.
[8] Ravishankar Ramamurthy David J DeWitt and Qi Su. 2002. A Case for Fractured Mirrors. In *Proceedings 2002 VLDB Conference: 28th International Conference on Very Large Databases (VLDB)*. Elsevier, 430.
[9] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1243–1254.
[10] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database–An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33.
[11] Christian Garcia-Arellano, Hamdi Roumani, Richard Sidle, Josh Tiefenbach, Kostas Rakopoulos, Imran Sayyid, Adam Storm, Ronald Barber, Fatma Ozcan, Daniel Zilio, et al. 2020. Db2 event store: a purpose-built IoT database engine. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3299–3312.
[12] Jana Giceva and Mohammad Sadoghi. 2019. Hybrid OLTP and OLAP.
[13] Anil K Goel, Jeffrey Pound, Nathan Auch, Peter Bumbulis, Scott MacLean, Franz Färber, Francis Gropengiesser, Christian Mathis, Thomas Bodner, and Wolfgang Lehner. 2015. Towards scalable real-time analytics: An architecture for scale-out of OLxP workloads. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1716–1727.
[14] Daniel Hieber and Gregor Grambow. [n.d.]. Hybrid Transactional and Analytical Processing Databases: A Systematic Literature Review. ([n. d.]).
[15] 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.
[16] J.S. Karlsson, A. Lal, C. Leung, and T. Pham. 2001. IBM DB2 Everyplace: a small footprint relational database system. In *Proceedings 17th International Conference on Data Engineering*. 230–232. https://doi.org/10.1109/ICDE.2001.914833
[17] 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.
[18] 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.
[19] 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.
[20] 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.
[21] 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.
[22] Barzan Mozafari, Jags Ramnarayan, Sudhir Menon, Yogesh Mahajan, Soubhik Chakraborty, Hemant Bhanawat, and Kishor Bachhav. 2017. SnappyData: A Unified Cluster for Streaming, Transactions and Interactice Analytics.. In *CIDR*.
[23] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 677–689.
[24] Oracle. 2021. Oracle Database 21c. https://docs.oracle.com/en/database/oracle/oracle-database/21/index.html.
[25] Fatma Özcan, Yuanyuan Tian, and Pinar Tözün. 2017. Hybrid transactional/analytical processing: A survey. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1771–1775.
[26] O'Neil Pat, O'Neil Betty, and Chen Xuedong. 2009. The Star Schema Benchmark.
[27] PostgreSQL. 2021. PostgreSQL Streaming Replication Documentation. https://www.postgresql.org/docs/current/warm-standby.html.
[28] PostgreSQL. 2021. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.
[29] PostgreSQL. 2021. Swarm64 HTAP Benchmark for PostgreSQL. (2021).
[30] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling highly contended OLTP workloads using fast dynamic partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.
[31] Iraklis Psaroudakis, Florian Wolf, Norman May, Thomas Neumann, Alexander Böhm, Anastasia Ailamaki, and Kai-Uwe Sattler. 2014. Scaling up mixed workloads: a battle of data freshness,fl exibility, and scheduling. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 97–112.
[32] 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.
[33] 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.
[34] Kun Ren, Jose M Faleiro, and Daniel J Abadi. 2016. Design principles for scaling multi-core oltp under high contention. In *Proceedings of the 2016 International Conference on Management of Data*. 1583–1598.
[35] Margy Ross and Ralph Kimball. 2013. *The data warehouse toolkit: the definitive guide to dimensional modeling*. John Wiley & Sons.
[36] Jimi Carmen Sanchez. 2016. Investigating the Star Schema Benchmark as a Replacement for the TPC-H Decision Support System. (2016).
[37] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 731–742.
[38] Alex Skidanov, Anders J. Papito, and Adam Prout. 2016. A column store engine for real-time streaming analytics. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1287–1297. https://doi.org/10.1109/ICDE.2016.7498332
[39] Michael Stonebraker. 1987. *The design of the Postgres storage system*. Technical Report. CALIFORNIA UNIV BERKELEY ELECTRONICS RESEARCH LAB.
[40] Michael Stonebraker and Lawrence A Rowe. 1986. The design of Postgres. *ACM Sigmod Record* 15, 2 (1986), 340–355.
[41] 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.
[42] Cong Yan and Alvin Cheung. 2016. Leveraging lock contention to improve OLTP application performance. *Proceedings of the VLDB Endowment* 9, 5 (2016), 444–455.
[43] 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.