TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads

Nuno Faria
INESCTEC and University of Minho
Braga, Portugal
nuno.f.faria@inesctec.pt

José Pereira
INESCTEC and University of Minho
Braga, Portugal
jop@di.uminho.pt

Ana Nunes Alonso
INESCTEC and University of Minho
Braga, Portugal
ana.n.alonso@inesctec.pt

Ricardo Vilaça
INESCTEC and University of Minho
Braga, Portugal
ricardo.p.vilaca@inesctec.pt

Yunus Koning
MonetDB Solutions
Amsterdam, The Netherlands
yunus.koning@monetdbsolutions.com

Niels Nes
MonetDB Solutions and CWI
Amsterdam, The Netherlands
niels.nes@monetdbsolutions.com
niels.nes@cwi.nl

ABSTRACT

Transactions have been a key issue in database management for a long time and there are a plethora of architectures and algorithms to support and implement them. The current state-of-the-art is focused on storage management and is tightly coupled with its design, leading, for instance, to the need for completely new engines to support new features such as Hybrid Transactional Analytical Processing (HTAP). We address this challenge with a proposal to implement transactional logic in a query language such as SQL. This means that our approach can be layered on existing analytical systems but that the retrieval of a transactional snapshot and the validation of update transactions runs in the server and can take advantage of advanced query execution capabilities of an optimizing query engine. We demonstrate our proposal, TiQuE, on MonetDB and obtain an average 500× improvement in transactional throughput while retaining good performance on analytical queries, making it competitive with the state-of-the-art HTAP systems.

PVLDB Reference Format:
Nuno Faria, José Pereira, Ana Nunes Alonso, Ricardo Vilaça, Yunus Koning, and Niels Nes. TiQuE: Improving the Transactional Performance of Analytical Systems for True Hybrid Workloads. PVLDB, 16(9): 2274 - 2288, 2023.
doi:10.14778/3598581.3598598

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

1 INTRODUCTION

Transactions in database systems allow sequences of operations, possibly issued interactively, to be grouped and made atomic regarding both concurrency and faults. The traditional architecture for a transactional system is that isolation and recovery are encapsulated within a Transaction Storage Manager (TSM) layer [61] using a form of locking or timestamp ordering to exclude anomalous non-isolated executions, and logging to guarantee atomic recovery and durability [40, 57]. Query processing happens on top of the abstraction of isolated and recoverable table and index storage exposed by the TSM. This architecture for a general-purpose database system does however impose a significant performance overhead [60] and has led to specialization [98]. This approach has been particularly successful for analytical workloads with the introduction of column-oriented systems, that enable advanced query processing techniques such as vectorized processing and adaptive indexing, however, at the expense of limited concurrent operational performance [27]. Figure 1 illustrates this tradeoff: MonetDB [8], in the lower-right corner, has much higher analytical performance than PostgreSQL [14], a general-purpose system, but its operational performance is limited and strongly impacted by concurrency. (Experimental conditions and further details in Section 4.)

Some scenarios call for high-performance analytical operations together with fine-grained update transactions to achieve Hybrid Transactional-Analytical Processing (HTAP). This is useful, for instance, for Internet-of-Things (IoT) and streaming applications, and has been a driver for recent innovation in transactional systems. The common way to do this is to start from a transactional system and add the ability to execute long-lived analytical queries on fresh

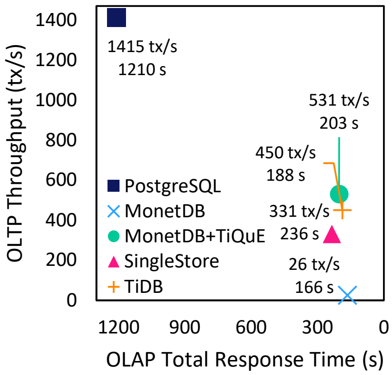


Figure 1: Trade-off between transactional and analytical performance and the contribution of TiQuE. Transactional and analytical workloads tested separately (32 OLTP clients; 1 OLAP client).

Proceedings of the VLDB Endowment, Vol. 16, No. 9 ISSN 2150-8097.
doi:10.14778/3598581.3598598

data by materializing a snapshot [34, 67], by outright replicating the database [63, 109], but also by implementing a storage manager that judiciously holds recent updates in memory, separately from persistent data [87]. As shown also in Figure 1 for TiDB and SingleStore, this allows for much higher operational throughput without compromising analytical performance.

Our goal is an alternative approach for implementing transactions that eases innovation in existing systems by avoiding the effort of rewriting the storage management layer. Our contribution is an add-on layer aimed at analytical systems. This should allow us to substantially improve transactional performance without costly redeployment and retraining processes for a new HTAP engine.

The key insight of our proposal – *transactions in the query engine* (TiQuE) – is to leverage the flexibility and efficiency of a generic query optimizer and execution engine in the manipulation of metadata (e.g., timestamps) required for multi-record transactions. This means that *part of the traditional responsibility of the TSM is actually expressed in a high-level query language such as SQL*. As shown in Figure 1, this allows us to build on MonetDB a solution that preserves a good analytical performance while competing with state-of-the-art HTAP in operational performance in a non-distributed deployment. As we show in Section 4.3.3, this approach is effective even with simultaneous operational and analytical workloads.

In addition to it being easier to deploy and maintain, this approach allows isolation criteria and concurrency control to be expressed as high-level queries. The latter point is especially important to tune performance based on workload [101] and client requirements (stronger, weaker, eventual, or none at all). It also allows transparent exploitation of advanced query processing techniques, such as vectorization, as found in column-oriented engines [27].

In short, the hypothesis supporting TiQuE is that *given the diversity and continuous evolution of application requirements, environments, and hardware capabilities, it is likely that a query optimizer (and tuning by developers and administrators) will generate a better (custom) implementation of multi-statement transactions than a single, one size fits all, hard-coded solution.* A particularly interesting consequence is that it can be applied to specific tables, avoiding any overhead on OLAP-only tables. Our main contributions are:

- We describe a set of assumptions on existing analytical data stores to support TiQuE, propose the mechanism in terms of a generic SQL schema and queries, and provide a correctness argument (Section 2)
- We specialize the generic proposal and describe an implementation of TiQuE on MonetDB, discussing the strategy used to ensure each assumption and to obtain the best performance (Section 3).
- We evaluate the correctness and performance of TiQuE as implemented on MonetDB, with demanding workloads, and compare it to state-of-the-art HTAP proposals (Section 4).

Finally, we discuss related work (Section 5) and conclude the paper, pointing out interesting future research directions (Section 6).

2 TRANSACTIONS IN THE QUERY ENGINE

Figure 2 summarizes the TiQuE approach. The Client application requests an Abstract schema that contains only application-level data. This is translated automatically by TiQuE to a Physical schema that exists in the database engine and contains also transactional metadata. Client requests are then intercepted by TiQuE, which translates them into queries that enforce isolation and forwards them to the Physical Layer for execution. We choose to target Snapshot Isolation [39], in which transactions execute under a data snapshot taken at their start time. This widely used isolation [1, 15, 21, 23, 25] is specifically chosen given it allows non-blocking reads with concurrent writes, ideal for HTAP workloads.

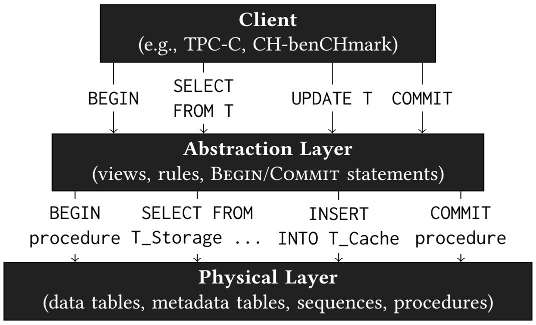

Client
(e.g., TPC-C, CH-benCHmark)
BEGIN
SELECT FROM T
UPDATE T
COMMIT
Abstraction Layer
(views, rules, BEGIN/COMMIT statements)
BEGIN procedure
SELECT FROM T_Storage ...
INSERT INTO T_Cache
COMMIT procedure
Physical Layer
(data tables, metadata tables, sequences, procedures)

Figure 2: Architecture of TiQuE.

2.1 Assumptions

The base assumption is that we have an optimizing query engine, that compiles and executes SQL statements, without[1] multi-item update transactions. In detail, we assume:

For generality. The generality of the approach, meaning that TiQuE can be used on different database systems and is applicable to existing off-the-shelf database applications, assumes that client requests can be intercepted and modified. For standard APIs, this can be achieved at the client side by a driver wrapper or hook. Alternatively, read statements can often also be intercepted at the server side by *views* and write statements by *rules*, using session variables to hold the transaction state. We use the first, more general approach in Section 3, although we have also tested the second.

For correctness. The correctness of the approach, meaning that TiQuE does, in fact, enforce Snapshot Isolation [39], depends on being able (i) to atomically and concurrently append rows to some tables, as well as being able to correctly read them; and (ii) to maintain and observe sequential counters. The first requirement is generally available, with high performance even in a distributed setting, in modern data stores. The second is not as widely available but can be implemented in user-defined functions (see Section 3) or resort to a coordination system such as ZooKeeper [64] in a distributed setting. In fact, some systems provide access to atomic counters even without multi-operation transactions [20].

For performance. The performance of the approach, meaning that TiQuE achieves high concurrent operational and analytical performance, depends on (i) the query processing engine being able to successfully optimize physical queries resulting from TiQuE and (ii) sustaining high throughput append-only tables. In Section 3, we show how both can be met by a state-of-the-art analytical engine.

[1] Or not making use of multi-item transactions, due to their impact on performance.

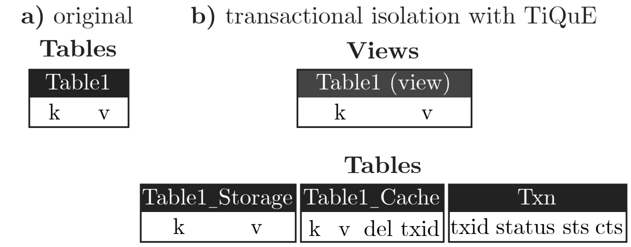


Figure 3: Example of an original schema (a)) converted into one which supports transactional isolation in SQL (b)).

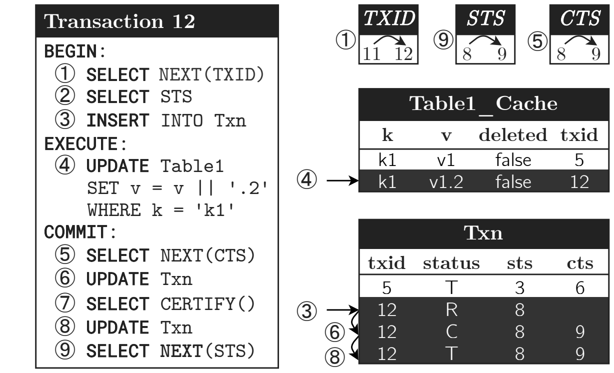
2.2 Schema translation

TiQuE uses regular tables to hold application data and transaction metadata, and SQL queries to manage both of them. As shown in Figure 3, for each Abstract Table defined by the application, TiQuE creates the corresponding Physical Schema to be used by future data query and modification operations. This is done as follows:

Storage tables. These tables contain data and have the exact schema requested by the application. We identify them in this text with the "_Storage" suffix. The rows in these tables are considered stable, meaning they are contained in the history of the snapshot of every current and future transaction. We use this property to our advantage to avoid storing timestamp metadata for each row.

Cache tables. These tables, identified with the "_Cache" suffix, contain additional data with associated metadata for each table requested by the application. Unlike storage tables, cache tables store uncommitted or recently committed data, and oftentimes store multiple versions of the same row [91]. As data stored in these tables might not be included in the history of every transaction, we append metadata to each row for snapshot computation (Figure 3): the *deleted* field – used to specify if a row was deleted (also known as a "tombstone") – and the transaction identifier (*txid*). This is analogous to what is done in the low-level representation of records by existing database systems, such as PostgreSQL [96, 99]. Eventually, cache data that becomes stable can be checkpointed to the respective storage table (Section 2.4.1). If the underlying data store supports it, we can place the cache tables in a different format from the storage ones (e.g., row format for cache and columnar format for storage), combining fast writes with fast analytical scans. Note that, for simplicity, we assume that a cache row referring to an Update also has the data of the columns that were not modified. An implementation of TiQuE could store only the modified columns and merge them with the previous data when reading. Also note that we consider cache tables to have the same schema as their storage counterparts (plus metadata). However, they could also be designed using, for example, a key-value model, where the value in each row is merged into a single column, later being deserialized when reading. This would still allow efficient selection by key.

Txn table. This table stores the transactions' metadata: the start (*sts*, for reading) and commit timestamps (*cts*, for conflict detection and write visibility) of every transaction executed or currently being executed, as well as their status, (*running* (R), *committing* (C), *committed* (T), or *aborted* (A)). These metadata are used to compute the snapshot, provide conflict detection, determine which data can be checkpointed to the storage, and enable recovery after a crash.



```
Transaction 12
BEGIN:
 ① SELECT NEXT(TXID)
 ② SELECT STS
 ③ INSERT INTO Txn
EXECUTE:
 ④ UPDATE Table1
    SET v = v || '.2'
    WHERE k = 'k1'
COMMIT:
 ⑤ SELECT NEXT(CTS)
 ⑥ UPDATE Txn
 ⑦ SELECT CERTIFY()
 ⑧ UPDATE Txn
 ⑨ SELECT NEXT(STS)
```

Table1_Cache

k	v	deleted	txid
k1	v1	false	5
k1	v1.2	false	12

Txn

txid	status	sts	cts
5	T	3	6
12	R	8	
12	C	8	9
12	T	8	9

Figure 4: Example of a transaction execution in TiQuE. Based on the schema presented in Figure 3. Note that steps ⑥ and ⑧ are updates to the previous *Txn* row and not new inserts.

2.3 Operation translation

2.3.1 Transaction demarcation. A transaction is started by executing a Begin procedure, which assigns transactions identifiers (*txid*) and starting timestamps (*sts*) (Figure 4 ①-③). Transactions in TiQuE execute over multi-versioned data and under optimistic assumptions, meaning they perform all of their reads and writes without acquiring any locks, and, at commit time, the write-set is validated against the write-set of concurrent transactions [71]. The optimistic model is widely used due to its low overhead in low-contention settings [22, 24, 46, 56, 107]. It also has the advantage of fitting nicely with TiQuE's design, as it does not require managing locks on writes, but can cause starvation of long-running read-write transactions. An optimization to this problem is later discussed and evaluated in Sections 2.3.4 and 4.3.4, respectively. By not requiring locking on writes, it also opens the possibility of buffering them at the client side [71], helping mitigate the effects of client-server latency on multi-write transactions.

To end a transaction, the Commit procedure is executed, committing or aborting it based on the certification outcome (Figure 4 ⑤-⑨). Read-only transactions can skip this step.

The following sections describe how reads (Section 2.3.2) and writes (Section 2.3.3) are translated in a way that is both efficient and transparent to existing application code, and how transactions are validated and committed (Section 2.3.4).

2.3.2 Snapshot computation. Read operations within TiQuE must ensure that an isolated snapshot is obtained independently of concurrent writes from other transactions. This is done by replacing each user-visible table with a computation of the snapshot in the context of the current transaction, by using a view or direct replacement in the intercepted statement. The resulting composite query is then compiled, optimized, and executed as a whole.

The transaction's starting timestamp (*sts*), assigned at its inception (Figure 4 ②), encompasses all previously committed transactions – that committed with timestamp *cts* and have a status of "*T*" – and as such is used to determine the data visible in the snapshot. If

Listing 1: Snapshot query of transaction MY_TXID.

```sql
1  SELECT k, v
2  FROM (
3    SELECT *, rank() OVER ( -- order recent versions first
4     PARTITION BY k ORDER BY cts DESC NULLS FIRST) AS rk
5    FROM (
6      -- Storage data
7      (SELECT k, v, false AS deleted, 0 AS cts
8      FROM Table1_Storage)
9      UNION ALL
10     -- Cache data
11     (SELECT k, v, deleted, cts
12     FROM Table1_Cache C
13     JOIN Txn ON Txn.txid = C.txid
14     -- get committed data or its own uncommitted writes
15     WHERE (Txn.status = 'T' OR Txn.txid = MY_TXID)
16     -- filter out future writes
17       AND (cts <= MY_STS or cts IS NULL))
18   ) T1
19 ) T2
20 -- select only the most recent version of each row
21 WHERE rk = 1
22   -- remove deleted rows
23   AND NOT deleted;
```

this timestamp captures more than one version of the same row, we only return the most recent one. Additionally, a transaction might itself have written uncommitted data which must be read back. These data will replace committed rows which have the same key, as they are considered more recent. With these considerations, the snapshot for *Table*1 of Figure 3 is computed with the SQL query shown in Listing 1, where MY_STS and MY_TXID are the session variables representing the transaction's *sts* and *txid*, respectively.

In detail, we first combine (line 9) storage data (lines 7,8) with readable committed data in the cache and the transaction's uncommitted writes (lines 11-17). Note that we project the storage's *cts* (which is not stored) as 0 (line 7), to guarantee that we order the stable rows after any other present in the cache. Additionally, we also project *false* as the *deleted* field in the storage (line 7), since we guarantee that deleted tuples are not persisted past the cache (Section 2.4.1). Next, since multiple past versions of the same row can be returned, we must select only the most recent one. To do so, we sort the rows by key and timestamp – prioritizing the transaction's writes – by performing a ranking window function (lines 3, 4) and selecting only the most recent version for each key, i.e., ranking of 1 (line 21). Finally, we filter out deleted rows (line 23).

Figure 5 illustrates the snapshot computation with an example. Briefly, the transaction's snapshot only contains rows $\langle k1, v100 \rangle$, $\langle k2, v20 \rangle$ and $\langle k4, v4 \rangle$. The remaining rows were either replaced by newer versions ($\langle k1, v1 \rangle$,$\langle k2, v2 \rangle$,$\langle k3, v3 \rangle$), were committed after transaction 4 started ($\langle k2, v2000 \rangle$), or were deleted ($\langle k3, \perp \rangle$).

Describing the snapshot reconstruction as a view has interesting consequences: First, when used in a complex query, the optimizer has the ability to translate it to different physical plans (Section 4.2). This admits the possibility that filters are pushed down below snapshot reconstruction and that the join operations used there are in fact re-ordered with other joins in the query. This can be particularly useful in the face of parallelism or vectorization. Second, there is also ample possibility of intervention, by an administrator, to configure indexes appropriate for each application scenario, or even switch to different view implementations.

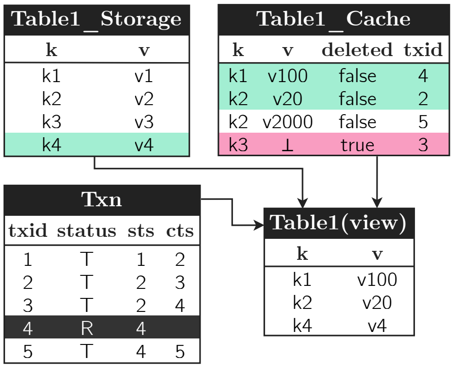


Table1_Storage

k	v
k1	v1
k2	v2
k3	v3
k4	v4

Table1_Cache

k	v	deleted	txid
k1	v100	false	4
k2	v20	false	2
k2	v2000	false	5
k3	⊥	true	3

Txn

txid	status	sts	cts
1	T	1	2
2	T	2	3
3	T	2	4
4	R	4	
5	T	4	5

Table1(view)

k	v
k1	v100
k2	v20
k4	v4

Figure 5: Example of the resulting snapshot of *Table*1 for transaction 4. $\langle k1, v1 \rangle$ is replaced by the transaction's temporary write $\langle k1, v100 \rangle$; $\langle k2, v2 \rangle$ is replaced by the committed tuple $\langle k2, v20 \rangle$; $\langle k2, v2000 \rangle$ was committed after transaction 4 started; $\langle k3, v3 \rangle$ is replaced by the committed $\langle k3, \perp \rangle$; $\langle k3, \perp \rangle$ is flagged as deleted.

One thing to note is that the code in Listing 1 is not the only way to compute the snapshot. Alternatives include: performing a left anti-join to filter the storage values with the cache; combining both tables with a full join instead of a union; using Distinct On instead of the window function; and so on. In practice, different snapshot reconstruction code can generate different plans. This is a key advantage of our solution, as we found that between multiple plans, one was optimal for transactional workloads while another was optimal for analytical ones (Section 3.2). For example, the snapshot in Listing 1 is not optimal for queries that retrieve large amounts of data as it requires computing a ranking window function over the returned rows. We can thus select the best query for each case.

2.3.3 *Handling writes.*

Insert, Update, and Delete statements in TiQuE must be rewritten to modify data according to its encoding of transactional metadata. All these statements will thus be translated into Inserts on the cache tables, to be later validated and committed. Figure 4 ④ shows an example where an Update to *Table1*, to concatenate ".2" to v of the row with k = "$k1$", is instead translated as an Insert to *Table1_Cache* of $\langle k1, v1.2, false, 12 \rangle$.

Listing 2 presents an example of an Update rule (PostgreSQL's syntax) on *Table*1 of Figure 3, where MY_TXID is the session variable that contains the transaction's identifier. In detail, we capture an Update to the view *Table1* (line 3) and redirect it as new Insert(s) to the respective cache, *Table1_Cache* (lines 6, 7). If the same transaction attempts to update the same row twice, we update the one previously inserted by it (lines 10, 11). Insert and Delete follow a similar implementation, with the only difference being that Delete inserts rows with the *deleted* flag set to true.

The flexibility of TiQuE, which enables slightly different implementations for each transaction, allows the option of buffering writes in the client just until the commit operation (Section 3.2.3), reducing round-trips. This precludes that a transaction reads its own writes but is admissible as many transactional applications do not in fact try to do it as they proceed in two separate steps: first, reading from existing data and then performing the updates. This is the case, for instance, in TPC-C [88]. This technique is also employed by data stores such as Spanner [49].

Listing 2: Definition of a rule over Updates (PostgreSQL).

```sql
CREATE RULE "update_rule" AS
  -- capture updates to view 'Table1' ...
  ON UPDATE TO Table1
  DO INSTEAD
    -- ... and perform new inserts to 'Table1_Cache'
    INSERT INTO Table1_Cache
      VALUES (NEW.k, NEW.v, FALSE, MY_TXID)
    -- replace previous inserts/updates/deletes
    -- performed by this transaction
    ON CONFLICT (k, txid)
    DO UPDATE SET v = EXCLUDED.v, deleted = FALSE;
```

Listing 3: Transaction validation for *Table1*.

```sql
1  SELECT EXISTS( -- returns 'true' if there are conflicts
2    SELECT 1
3    FROM ( -- select the transaction's write-set
4      SELECT k
5      FROM Table1_Cache
6      WHERE txid = MY_TXID
7    ) WS
8    -- join by primary key to find potential conflicts
9    JOIN Table1_Cache C ON C.k = WS.k
10   -- get the metadata of the potential conflicts
11   JOIN Txn ON Txn.txid = C.txid
12   -- it's a conflict if it ended during this txn's runtime
13   WHERE Txn.cts > MY_STS and Txn.cts < MY_CTS
14     -- and committed or currently committing
15     AND (Txn.status = 'T' OR Txn.status = 'C')
16 );
```

2.3.4 Certification and commit. Our Commit procedure can be divided into three main steps: obtaining the commit timestamp; performing write-set certification, in which the transaction's writes are compared against all non-stable data committed after it started; and durable confirmation of the transaction as committed or aborted.

The first step uses a monotonically increasing counter to assign a commit timestamp (*cts*), stored in the transaction's record on the Txn table (Figure 4 ⑤,⑥). Note that we use two separate sequences to generate the commit and starting timestamps. Although a single one would suffice for correction, it would block starting transactions while another is committing, to prevent reads of uncommitted data.

In the second, certification, we validate that a transaction T can commit if there were no write-write conflicts with concurrent transactions (Figure 4 ⑦). In this case, it means not having a new version of some key in the write-set that was committed after T started – i.e., with a *cts* greater than its *sts*. Given that we store all data and metadata using relations, the conflict detection algorithm is also computed using SQL code. Using $Table1$ of Figure 3, the existence of conflicts can be computed with a join between T's write-set and the previously committed/currently committing data, as exemplified in Listing 3, where MY_TXID, MY_CTS, and MY_STS are session variables referring to T's identifier, *cts*, and *sts*, respectively. In detail, we determine T's write-set (lines 3-7), find potential conflicts by joining by primary key (line 9), determine the potential conflicts' metadata (line 11), and select only the data of the transactions that ended during the runtime of T (line 13) and either committed or are committing (line 15). If the certification step for some table returned any conflicts, we consider T as aborted. If not, we repeat the process for the other tables. If no conflicts are found for any table, we consider T as valid.

Note that by formulating the certification step as a SQL query, it becomes eligible for optimization. For instance, in a distributed-parallel system where data are partitioned by the application key column, k, execution of this step should naturally be executed in parallel. Also note that the storage tables are not used in the certification step, as data there is stable and therefore is guaranteed to be committed after the transaction started, improving performance.

For the third and final step – i.e., mark the transaction as committed or aborted – we only need to set its status in *Txn* as "*T*" or "*A*", respectively (Figure 4 ⑧). As this is an atomic operation, transactions in TiQuE are therefore also atomic. The current *sts* needs to be updated to make the newly committed data visible in the snapshot of future transactions. As multiple transactions can commit concurrently, we must ensure that *sts* advances monotonically and only after all transactions with $cts \leq sts$ have finished. To do this, a transaction T simply needs to wait until $sts = cts_T - 1$ and then setting *sts* to cts_T (Figure 4 ⑨). Since this can cause transactions with long write-sets to delay other concurrent transactions, one alternative would be to decouple the commit ordering from the commit timestamp. However, we consider here only a single timestamp to more easily describe our solution.

The flexibility of TiQuE also allows mitigating the possibility of starvation due to long-running read-write transactions and the first-committer-wins rule [71] with the option of assigning transactions a *priority* flag: on a failed certification, instead of marking the transaction as aborted, we refresh its starting timestamp and re-execute it. Assuming that the key-set remains the same between tries, the long-running transaction aborts, at most, once. We explore this possibility in Section 4.3.4.

2.4 Checkpointing and recovery

2.4.1 Checkpointing. As TiQuE relies on storing multiple versions of the same row, we employ a periodic checkpointing mechanism that 1) moves the stable cache data to the storage and 2) removes obsolete rows. This prevents large data storage overheads and keeps the cache tables relatively small.

Stable data refers to the rows which have $cts \leq \lambda$, where λ is the minimum *sts* of the set of running transactions, or the database's current *sts* if none is running, meaning they are included in the history of every current and future transaction. When flushing the stable cache data to the storage, we simply overwrite old versions, as they are now obsolete. Likewise, we only flush the most recent version of each stable row to the storage, as the others are not readable anymore. If the most recent version of a row was marked as deleted, we also do not flush it to the storage, in addition to deleting the storage version (if present). After flushing, we delete the stable rows from the cache. In addition to the cache tables, the stable timestamp is also used to clean up the *Txn* metadata table.

Checkpointing does not require atomicity, therefore, we can flush one row at a time and still ensure consistent snapshots. The reason for this is that data being flushed will always be read first from the cache, replacing storage versions currently being modified, thus precluding concurrency issues. Likewise, deleting stable cache data does not require atomicity, as long as we delete them by ascending *cts* order (another alternative is to first delete the older versions from the cache tables and then delete the most recent ones). Although we can end up with a snapshot that contains, for the same transaction, half of the data retrieved from the storage and half retrieved from the cache, it will always contain completed transactions – as stable cache data being deleted is guaranteed to be persisted in the storage – and the view will prevent duplicate versions. This property is important since it means that the checkpointing operation has minimal impact on foreground load, as it does not require blocking running transactions.

2.4.2 Recovery. The recovery process procedure runs upon restart and relies on *Txn* table, working as follows: The first thing we do is set the running ($status$ = "R") and committing ($status$ = "C") transactions as aborted. Although committing transactions could be salvaged, this simplifies the recovery and ensures the client that their uncommitted transactions always abort in case of a crash; Then, we set the sts value to the last committed cts. This ensures that the committed transactions that were waiting to advance the sts are visible in the snapshot (to handle crashes between ⑧ and ⑨ of Figure 4); Finally, we also set the current cts to the last committed cts. This ensures that future transactions are able to advance, as they must wait for the previous cts to be visible in Txn in order to proceed (to handle crashes between ⑤ and ⑥ of Figure 4).

2.5 Correctness argument

We argue that multi-row transactional execution under Snapshot Isolation can be correctly ensured with two simple guarantees: 1) atomic single-row writes respecting total order and 2) non-corrupt single-row reads. Next, we provide the reasoning behind this assumption, which is backed up by empirical testing in Section 4.1.

Atomic multi-row transactions in TiQuE mimic the ones found in data stores using *write-ahead logging* (WAL) [81]: writes, together with the respective identifier, are added to append-only mediums (*Cache* tables), while the transaction commit (or abort) is marked at the end with another write (*Txn* table). As the commit marker is a single row with the identifier, status, and timestamps, its write is atomic according to 1), thus the transaction is by consequence also atomic. Just like WAL, data from transactions without the respective commit marker are not recovered after a crash (Section 2.4.2).

Snapshot Isolation requires a transaction T_i to be able to read the most recent committed data if and only if it belongs to a transaction T_j ($T_i \neq T_j$) committed before T_i started. TiQuE ensures this by assigning transactions monotonically increasing start (sts) and commit timestamps (cts). When some T_j finishes, the current sts is incremented – with total order guarantees – to include $T_j.cts$ (Section 2.3.4). When T_i starts, the $T_i.sts$ provided is thus guaranteed to include only data committed before that time (Listing 1, l. 17). This also avoids non-repeatable reads, as future commits are not included. Likewise, dirty reads are also ignored, as data from a non-committed T_j are filtered out by not having the commit marker (Listing 1, l. 15). As for the most recent data requirement, this is ensured by sorting the readable versions and selecting the first one (Listing 1, l. 3, 4, 21), which also precludes primary key constraint violations. These points rely on the ability to read uncorrupted data from the *Cache*, *Storage*, and *Txn* tables, which is guaranteed by 2).

Finally, Snapshot Isolation states that T_i can commit if and only if there is no other T_j committed in T_i's interval and wrote data that T_i also wrote. This is ensured by intersecting T_i's key-set (Listing 3, l. 3-9) with data whose $cts \leq T_i.sts$ (Listing 3, l. 13) while ignoring data committing after $T_i.cts$, which conforms with the Snapshot Isolation's first-committer-wins rule.

Listing 4: Changes made to the reads and writes in application code running under MonetDB with TiQuE (Python syntax).

```
# database schema
schema = { ... }
# dynamic snapshot generation
snapshot = lambda tablename, sts: f""" (
  SELECT {columns(tablename, join=',')} -- e.g., k, v
  FROM ... ) as {tablename}_snapshot """
# reads (SELECT * FROM Table1 WHERE k <= 10)
cursor.execute(f"""SELECT *
  FROM {snapshot('Table1', sts)} WHERE k <= 10""")
# writes (UPDATE Table1 SET v = v+1 WHERE k <= 10)
cursor.execute(f"""
  INSERT INTO Table1_Cache SELECT k, v + 1, false, {cts}
  FROM {snapshot('Table1', sts)} WHERE k <= 10""")
```

3 IMPLEMENTATION

We implement TiQuE on MonetDB, which focuses on analytical performance by employing a column-based storage architecture and vectorized execution, among other design decisions [65].

3.1 Meeting requirements

3.1.1 Views, rules, and session variables. The implementation of TiQuE in MonetDB is complicated by not having support for *session variables* or *rules*. To work around this, we implement reads by dynamically building the SQL queries with the identifier injected into them. As for writes, we also manually convert them into new inserts to the respective cache tables. Listing 4 shows an example of the modifications we perform for the reads and writes.

3.1.2 Avoiding Update. As MonetDB does not support Updates on the same column without triggering concurrency-induced rollbacks, using a single *Txn* table and performing successive Updates was not viable. Therefore, we split this table into one for each transaction status (*began*, *committing*, *committed*, and *aborted*). This way, we only rely on the Insert statement, which does not trigger conflicts.

3.1.3 Sequences. As the snapshot must advance sequentially, a transaction needs to wait for the current sts to be equal to its $cts - 1$. Our initial implementation simply relied on a MonetDB sequence and actively waiting for the current sts to match $cts - 1$, which meant successively querying the database. As expected, the active wait, combined with the successive Selects, would lower scalability. Our current solution addresses this by relying on MonetDB's powerful *user defined functions* [10] – in C – to generate sequences and waiting using *pthread* conditions.

3.1.4 Unlogged tables. As most metadata (apart from the transaction identifier and the commit timestamp) are not required after the transactions that depend on them finish, we can rely on fast, in-memory tables to store them. Given that MonetDB's temporary tables are constrained to the session they were created, we implemented *unlogged* tables that offer the same functionality as regular tables but without flushing to disk, meaning data is not persisted on server restarts but write performance is greatly increased.

3.1.5 Begin and Commit. As MonetDB, much like other systems such as PostgreSQL, executes stored functions in a transactional block, we had to rely on application code to implement the Commit procedure, as otherwise we would not be able to read the most recent *Txn* data while committing, which is critical for TiQuE. Nonetheless, even if we had non-transactional procedures, we still need to call `begin()` and `commit()` explicitly in the application code, which is not ideal. Again, just like with reads and writes, this can be avoided with a driver-level implementation.

3.2 Performance tuning

Finally, we consider tuning the combined application and TiQuE workload. This is a key advantage of our proposal, as it allows database system developers and administrators to tune the performance of the transactional layer using common techniques and tools. To make it a fairer comparison, native MonetDB also relies on the optimizations described in Sections 3.2.1, 3.2.3, and 3.2.5.

3.2.1 Optimizer pipeline. The first thing we noticed was the fact that our snapshot computation was considerably slower than the native Select. Although we expect some overhead, our solution was more than 10× slower than the baseline. We narrowed the problem down to the Union All operator, as performing it with two simple Selects resulted in more than 9× the response time than running a simple Select on its own. The cause was the considerably large plans generated for queries with a Union All operator. After testing various MonetDB optimizer pipelines [9], we found that the `minimal_fast` yields the best results, making the Union All just 1.5× slower than the single Select.

3.2.2 OLTP and OLAP snapshots. We found the snapshot code in Listing 1 to be the best alternative for transactional queries, i.e., queries that often retrieve only one or a few rows. However, using it with analytical queries would not produce a good plan, even leading to out-of-memory crashes. For those queries, we take advantage of the possibility to use different formulations for the same query and use a snapshot that separately processes each table and combines the results, avoiding full-relation sorts.

3.2.3 Buffered writes. We also buffer a transaction's writes in the client and, at commit time, send them to the database to be applied. This allows multiple writes in the same TiQuE transaction to be flushed to disk simultaneously, which would otherwise not be possible as we rely on the auto-committed mode and cannot control the flush manually. The downside is that a transaction is not able to read the effect of its temporary writes. However, that is often not needed, as is the case with TPC-C, thus, this is often done in transactional systems [49].

3.2.4 Materialized write-set relation. To improve performance, we use only a single in-memory relation to certify transactions, named *Write_Sets*. This relation is comprised of two columns, one containing the transaction identifier and another containing the respective table name (shortened) and primary key hashed as a single column (e.g., for an update to the TPC-C's *Order_line* table, we might have $\langle 12, hash(OL.4.2.1234)\rangle$). While hashing reduces the cost of conflict detection, it can lead to false conflicts, albeit with an incredibly small probability with a good hash function as there are 2^{64} combinations. Instead of inserting the write-set in the respective cache tables, we first insert the write-set keys into the *Write_Sets* table, which is considerably faster. Now, as the entire write-set is contained in a single table, we only need to perform the query in Listing 3 for a single table, ensuring a faster certification.

3.2.5 Multi-column indexes. Next, still in the context of the read performance, we found that filtering data by multiple primary key columns was more expensive than filtering by just one. Although MonetDB supports multi-column indexes, they are not quite optimized for multiple, low-cardinality columns. It so happens that most TPC-C data are partitioned by *warehouse* and *district*, which are repeated numerous times. While a future update might improve this, for now, we rely on extra columns that contain the concatenation of the primary-key columns of their respective tables. This reduces response time but incurs a larger storage overhead.

4 EVALUATION

We evaluate TiQuE qualitatively, by testing that it correctly enforces Snapshot Isolation and that it indeed takes advantage of different execution plans in different settings. We then evaluate it quantitatively by measuring its performance when compared to baseline, OLAP-first MonetDB, general-purpose PostgreSQL, and state-of-the-art HTAP SingleStore and TiDB.

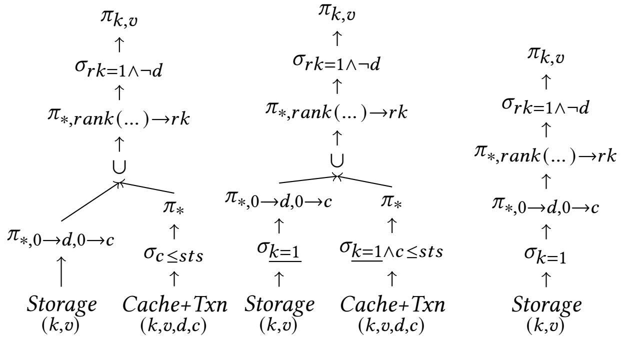
4.1 Correctness evaluation

We validate the correction of TiQuE's Snapshot Isolation by using the Elle transactional consistency checker [2, 33], developed and used by Jepsen in their database analyses [6]. Elle generates a workload with read and write operations on a key-value schema, where the values are lists that receive successive appends. This type of workload makes each key contain its entire update history, allowing Elle the strongest inference rules. We then execute the workload against TiQuE running over MonetDB, with auto-commit enabled to not rely on MonetDB's own isolation. Finally, the transactions' results (reads and commit/abort outcome) are tagged with the wall clock time and fed back to Elle, which performs inference analysis.

After several executions of 100k transactions and 8 clients, Elle always returned `{:valid? true}`, meaning TiQuE passed the Snapshot Isolation consistency checks. Elle did report `G2-item` anomalies [28], also known as write-skews on disjoint read, but these are expected, as Snapshot Isolation does not preclude them [39]. To test the checker, if we do not use the transaction's *sts* for reading but instead rely on the most recent one, it returns the anomalies `G0`, `G1c`, `G-nonadjacent`, and `G-single`, all not allowed by Snapshot Isolation. Likewise, if we remove conflict detection and instead commit all transactions, it also fails.

4.2 Execution plans

A key aspect of TiQuE, that demonstrates its usefulness beyond simply being easier to develop or deploy, is the possibility of the query planner optimizing the snapshot computation based on the context it is used on. For instance, as OLTP workloads often rely on reads of just one or a few rows, being able to push the filter(s) used to the source tables reduces the amount of data materialized and exploits underlying mechanisms such as indexes.


$\pi_{k,v}$
$\sigma_{rk=1\wedge\neg d}$
$\pi_{*,rank(\dots)\rightarrow rk}$
$\cup$
$\pi_{*,0\rightarrow d,0\rightarrow c}$
π_{*}
$\sigma_{c\leq sts}$
Storage
(k,v)
Cache+Txn
(k,v,d,c)
$\pi_{k,v}$
$\sigma_{rk=1\wedge\neg d}$
$\pi_{*,rank(\dots)\rightarrow rk}$
$\cup$
$\pi_{*,0\rightarrow d,0\rightarrow c}$
π_{*}
$\sigma_{\underline{k=1}}$
$\sigma_{\underline{k=1}\wedge c\leq sts}$
Storage
(k,v)
Cache+Txn
(k,v,d,c)
$\pi_{k,v}$
$\sigma_{rk=1\wedge\neg d}$
$\pi_{*,rank(\dots)\rightarrow rk}$
$\pi_{*,0\rightarrow d,0\rightarrow c}$
$\sigma_{k=1}$
Storage
(k,v)

(a) Regular snapshot **(b) a) + filter $k = 1$** **(c) b) + empty *Cache***

Figure 6: Demonstration of how the logical snapshot plan is optimized based on the query and data involved in MonetDB. Based on the snapshot of Listing 1 and schema of Figure 3. The *Cache* join with *Txn* is omitted to simplify the plans. d is the *deleted* column, c is the *cts* column, and sts is the transaction's start timestamp.

Figure 6a sketches the logical, relational algebra [45] plan based on the snapshot computation of Listing 1 using the buffered write-set optimization, meaning the *Cache* does not store temporary data. To filter the snapshot by $k = 1$, a naive implementation would simply apply the selection after the entire snapshot is computed, leading to the materialization of both tables. Instead, MonetDB optimizes the query and pushes $k = 1$ directly to both source tables, as displayed by the plan in Figure 6b. In detail, $k = 1$ is pushed down before the projection in the *Storage* and appended to the existing selection in the *Cache*. This means that, considering k is unique, the result of the union will end up materializing, at most, two rows, which also reduces the complexity of the window ranking function.

MonetDB also supports a cost-based optimization model to improve its query planning. This means that it is able to, for example, completely prune a branch that is expected to not return any data. This is particularly desirable for TiQuE as it allows to remove the *Cache* table from the plan if it does not contain any rows, as illustrated in Figure 6c. A future improvement could further remove the extra (unnecessary) computation done over the *Storage* table, resulting in the same performance as the native read in this case.

4.3 Performance

In this section, we compare the transactional and analytical performance of MonetDB with TiQuE against native MonetDB, PostgreSQL,[2] SingleStore,[3] and TiDB.[4] With native MonetDB, we will evaluate the effective overhead and performance gain of implementing transactions with the query engine. With PostgreSQL 14, we will evaluate how TiQuE in an OLAP-first database stacks up against a traditional row-based, OLTP-first database, in both transactional and analytical workloads. We chose PostgreSQL because it offers not only fast transactional performance, but is also able to parallelize scans, aggregations, sorts, and even joins [16]. This makes it a fairer comparison for analytical workloads compared to using traditional database systems with limited or no parallelism, such as MySQL 8. Finally, we use SingleStore 8.0.4 and TiDB 6.5.0 to evaluate how TiQuE compares to state-of-the-art HTAP systems. While TiDB ensures HTAP by keeping two copies of the data, asynchronously replicating from the row store to the column store, SingleStore uses a column store as the main storage and in-memory row storage for recently modified data.

We manually optimized all systems independently for transactional (TPC-C) and analytical (CH-benCHmark) workloads according to best practices, such as creating indexes or optimizing system parameters.[5] One particular parameter optimization worth pointing out is that we changed TiDB's default execution mode from pessimistic to optimistic, as we found that the latter would achieve higher transactional throughput (around 9%) in low-contention [24]. We also converted, in all systems, the *floats* to *decimals*, i.e., from floating point to fixed precision. The reason is that MonetDB does not parallelize aggregations with floating points, as computations in different orders – inherent from parallel executions – can return different results. In contrast, both TiDB and SingleStore parallelize numerical computations independently of the underlying precision.

The transactional (Section 4.3.1), analytical (Section 4.3.2), and long-running tests (Section 4.3.4) use a Google Cloud instance with 32 vCPUs (N1 Series), 32 GB RAM (with swap enabled), and 500 GB SSD. For the HTAP tests (Section 4.3.3) we also consider an instance with extra memory (128 GB). All tests run over a TPC-C dataset of 512 warehouses, which ensures the database does not fit entirely into the memory of the 32 GB RAM instance.

4.3.1 *Transactional workloads.*

We use the TPC-C workload [88] to evaluate the overall transactional performance, namely, the `py-tpcc` implementation.[6] We run different numbers of clients against each system, each with a duration of one minute. Each client can access any warehouse, executing a transaction as soon as the previous one finishes. We populate the database once at the start, after which we execute a warmup run. Figures 7a and 7b show the obtained throughput and abort rate, respectively.

Comparing first MonetDB with and without TiQuE shows the advantages of using an efficient transactional manager. In terms of throughput (Figure 7a), MonetDB with TiQuE reaches close to 1000× than the version without (average of 527×). As row-level granularity is provided, TiQuE avoids false conflicts, leading to more useful work being performed. On the other hand, the native MonetDB's abort rate is already at 49% with just 2 clients (Figure 7b).

Despite being the main limitation, false conflicts are not the only one. MonetDB relies on auxiliary structures for fast performance, such as hashes, that might be recomputed if there are modifications to the table. Although MonetDB reuses these hashes on a best-effort basis (meaning a single write should not require its recalculation), they might need to be recomputed, and the larger the table, the more expensive the recomputations. This is not an issue for TiQuE, as it redirects writes to the cache tables which are kept relatively smaller, and writes to the storage tables are periodic. This is why TiQuE with a single client ends up beating native MonetDB.[7]

[2] In PostgreSQL we use Repeatable Read, which is Snapshot Isolation in practice [15].

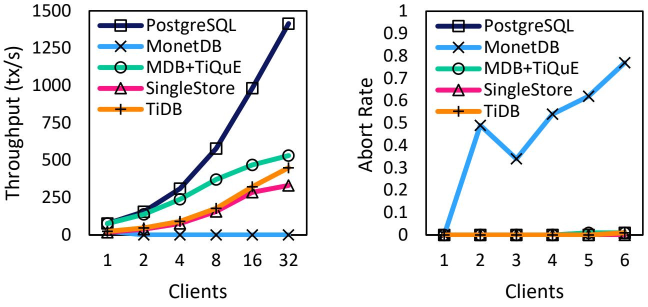
[3] SingleStore only supports the Read Committed isolation level [26]

[4] In TiDB we use Repeatable Read, which is also Snapshot Isolation [25].

[5] All configurations and their justifications are included in the companion artifact.

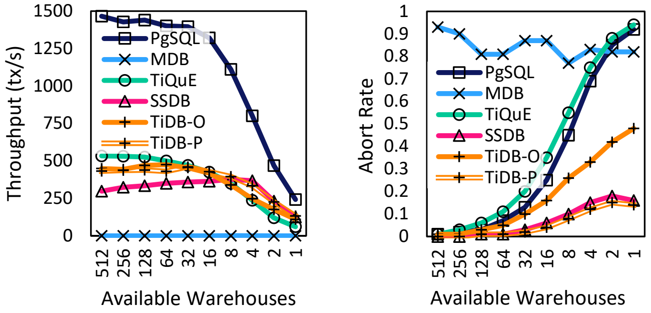
[6] https://github.com/apavlo/py-tpcc

[7] This only happens with relatively large datasets. With a smaller one (e.g., 1 warehouse instead of 512) native MonetDB's throughput beats TiQuE's when using one client.



(a) Throughput **(b) Abort rate**

Figure 7: Performance comparison of TiQuE in a transactional workload (TPC-C).



(a) Throughput **(b) Abort rate**

Figure 8: Performance comparison of TiQuE in a workload with variable contention (TPC-C with 32 clients). The x-axis dictates the range of warehouses that can be used by 99% of the transactions.

As for PostgreSQL, its throughput is on average 1.6× higher than MonetDB with TiQuE (up to 2.7×). One of the main reasons is that row storage outperforms column storage for simple reads and writes [48]. We empirically found that MonetDB's point reads and writes are more than 300% and 50% slower than PostgreSQL's, respectively. However, TiQuE managed to considerably reduce the difference in throughput between PostgreSQL and MonetDB, so we see this as truly positive results.

Another positive result is that TiQuE manages to be competitive in transactional workloads with SingleStore and TiDB HTAP systems, even surpassing them. As these are distributed-first systems (although using single-node deployments) with emphasis on scalability, it is expected some performance loss when compared to TiQuE with a single-node data store such as MonetDB.

The second set of tests evaluates TiQuE under variable contention. Variable contention with the TPC-C workload is done by restricting the number of warehouses that can be accessed. The results, plotted in Figure 8, start with all the warehouses available (512) and exponentially limit the number to just one. The number of clients is fixed at 32. To keep all data being used, 1% of the transactions are redirected to any warehouse. We evaluate both TiDB with the optimistic (`TiDB-O`) and pessimistic (`TiDB-P`) modes.

The throughput data in Figure 8a shows that contention increase in TiQuE is indirectly correlated with performance, due to the higher abort rate (Figure 8b), with the drop in performance becoming significant after 16 available warehouses (or after an abort probability of $\frac{1}{3}$). TiQuE and TiDB-Optimistic show similar results, as both execute transactions optimistically, meaning more work is wasted with more collisions. TiDB-Pessimistic, on the other hand, relies on locking, which results in faster performance in high contention. In this mode, TiDB first waits for the lock to be released and then advances, thus only rollbacking on primary key violations. Likewise, PostgreSQL also relies on locking for writes, but instead of waiting and advancing after the lock is released, it waits and aborts the transaction if the original lock holder succeeds. SingleStore, which only provides the Read Committed isolation level, takes longer to be affected by the high collision probability. Native MonetDB maintains the same performance throughout, as its conflict detection is column-based.

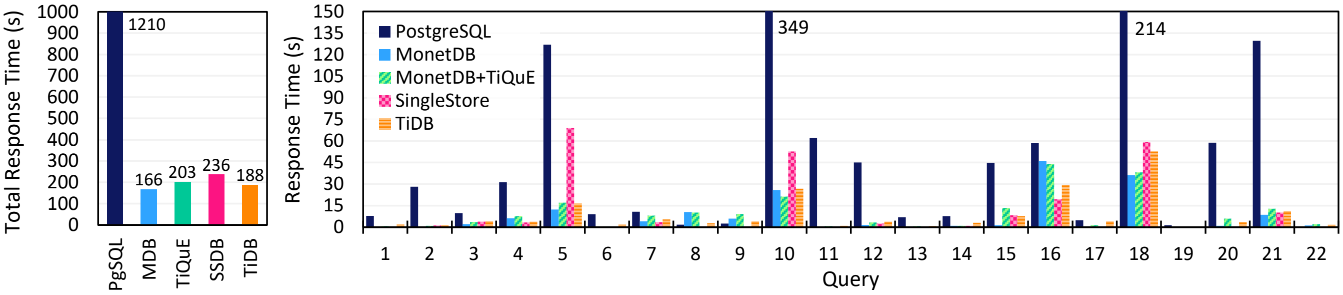
4.3.2 Analytical workloads. We now evaluate the performance of TiQuE in analytical workloads, namely with CH-benCHmark [47], which performs queries similar to TPC-H over TPC-C data. We run each query individually five times, sequentially, averaging the response time. The database is populated once at the beginning, and an OLTP run is executed to fill TiQuE's cache tables. Figure 9a displays the total average response time and Figure 9b the average response time for each query.

Starting again by comparing both MonetDB versions, we see that they display similar response times. Overall, TiQuE on MonetDB increases response time by around 20%. Even though TiQuE's snapshot computation requires joining more tables and filtering by timestamp, MonetDB is able to optimize them well enough. This shows the viability of TiQuE, which is able to significantly increase transactional performance with little impact on analytical workloads, proving our initial thesis.

SingleStore and TiDB, both column stores, also report similar results to TiQuE and MonetDB. Overall, TiQuE is 8% slower than TiDB and 16% faster than SingleStore, demonstrating TiQuE is also competitive with HTAP systems in analytical workloads.

PostgreSQL is, on the other hand, noticeably slower than the columnar systems, being on average, 6× slower than TiQuE. Although it is also able to exploit parallelism, it will be bounded by data transfers between disk, memory, and CPU cache. The column store engines, on the other hand, will retrieve considerably less data, since they can exploit the fact that most queries will only need a small subset of a table's columns at a time. Furthermore, in MonetDB specifically, the columnar representation is not only applied to the storage layer but also to the in-memory representation, by using memory-mapped files. As data is represented by consecutive C arrays of compressed primitive values, MonetDB also maximizes data cache locality. This representation is also used to exploit vectorization features offered by most CPUs, further improving performance [65].

4.3.3 Hybrid workloads. Figure 1 plots the analytical response time against the best transactional throughput of each system, according to the results of Sections 4.3.2 and 4.3.1, respectively. We conclude that, with both workloads running separately, TiQuE achieves a balanced tradeoff between transactional throughput and analytical



(a) Total time

(b) Time by query

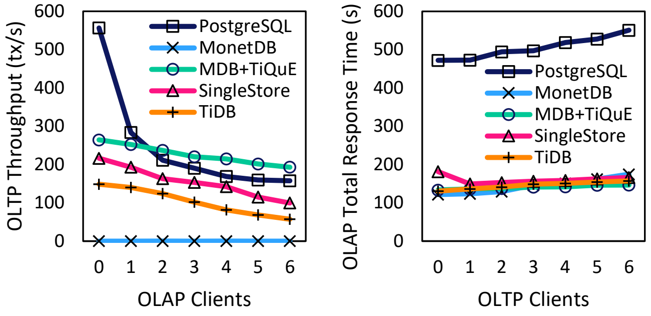
Figure 9: Performance comparison of TiQuE in an analytical workload (CH-benCHmark).

response time, consistent with HTAP systems, while PostgreSQL and native MonetDB only excel at OLTP or OLAP, respectively.

The next experiments evaluate TiQuE with concurrent OLTP and OLAP workloads. In theory, TiQuE's design should make the transactional throughput behave the same, while the response time of analytical workloads should slightly increase due to the snapshot having to join more data. To measure this, we perform two types of tests. The first one runs 6 fixed OLTP clients against a variable number of OLAP clients – 0 to 6 – measuring the impact of analytical load on OLTP (3 minutes for each test). Each analytical client is fixed to use at most 4 threads, meaning, at most, the system uses 30 of 32 threads (6 OLTP and 24 OLAP), avoiding CPU contention. The second runs 6 fixed OLAP clients with 0 to 6 OLTP ones, measuring the impact of transactional load on OLAP (the test ends when the 22 analytical queries have been executed). We use the 128GB RAM instance for these tests, as this was found to be a limiting factor for all systems with additional concurrency.

In Figure 10a, we observe that TiQuE's OLTP throughput suffers a relatively small decrease. Although the workloads do not contend for either computation or memory, they still have to share the L3 cache, which is small (45MB) compared to the amounts of data being processed. Additionally, internal locking in MonetDB to ensure data consistency also plays a part in the overall performance. PostgreSQL, on the other hand, takes a hit of 50% in throughput with the addition of one background analytical worker, descending to around 28% of the original throughput with the 6 OLAP clients. After 2 OLAP clients, PostgreSQL's OLTP throughput ends up matching TiQuE's. Conversely, native MonetDB's throughput stays close to zero throughout the test, due to its aforementioned challenges when dealing with transactional workloads. Both SingleStore and TiDB behave similarly to each other, slowly decreasing operational throughput with each new analytical background worker.

The second test (Figure 10b) shows that TiQuE's OLAP performance remains mostly constant with the increasing OLTP load, with a minor increase in response time. Native MonetDB, although starting with a faster execution than TiQuE, converges with it after 3 OLTP clients, as frequent direct updates on the base tables start to impact analytical execution. PostgreSQL's OLAP response time, which is significantly higher, slowly increases with the growing background transactional load. SingleStore and TiDB evolve comparatively with TiQuE, with TiDB starting slightly faster, which is in accordance with the analytical results of Figure 9.



(a) OLAP on OLTP

(b) OLTP on OLAP

Figure 10: Impact of background OLAP load on OLTP and background OLTP load on OLAP, using 128 GB of memory (TPC-C and CH-benCHmark). a) and b) run with 6 fixed OLTP and OLAP clients, respectively.

Both tests in Figure 10 let us conclude that, with sufficient hardware resources, OLTP and OLAP workloads in TiQuE have relatively little impact on each other's performance. With enough load, TiQuE ends up surpassing native MonetDB and PostgreSQL in both workloads, proving to be a good option in HTAP.

To summarize, we execute a test where we scale both workloads concurrently, using 1, 2, 4, 8, and 16 OLTP and OLAP clients (128 GB RAM). We set the number of analytical threads to 1, effectively splitting in half the available CPU resources.[8] Each number of clients runs until all 22 OLAP queries have been completed.

The results in Figure 11, which plot the OLTP throughput based on the total OLAP response time, show that TiQuE is able to scale both workloads executing concurrently, competitive with HTAP systems. PostgreSQL, on the other hand, displays an overall faster transactional throughput but its analytical performance stagnates and significantly declines after 8 clients. MonetDB exhibits an overall good analytical response time but a low transactional throughput. At the maximum load, TiQuE is 7.8× faster than PostgreSQL at OLAP, even surpassing native MonetDB's analytical performance, while being only 1.16× slower than PostgreSQL at OLTP.

[8] TiDB with 1 thread aborts analytical query 16 halfway through its execution, due to a data transfer limit (see https://github.com/pingcap/tiflash/issues/3436).

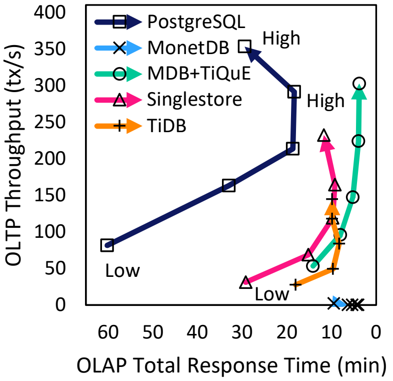
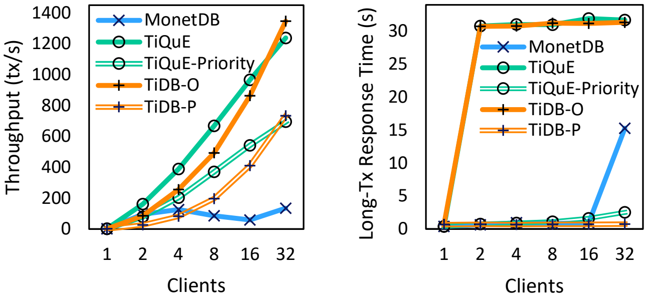


Figure 11: Performance comparison of TiQuE in HTAP (TPC-C and CH-benCHmark). Each test starts with 1 OLTP and 1 OLAP clients (Low) and ends with 16 OLTP and 16 OLAP clients (High).



(a) Throughput

(b) Long-tx response time

Figure 12: Performance comparison of TiQuE in a simple operational workload with long-running transactions.

4.3.4 Long-running read-write transactions. The final set of tests evaluates the performance of long-running, read-write transactions (e.g., data-cleaning) in the current TiQuE implementation, which executes optimistically. Since the first committer wins, long-running transactions can successively rollback due to conflicts with shorter ones. Our theory is that the priority flag provided by TiQuE (Section 2.3.4) can mitigate this issue. To do so, we built a simple benchmark that executes simple read-write transactions (half reads, half writes) with a variable number of clients (each test runs for 30 seconds; transactions are retried if aborted), on a table with 10k rows. Of the n clients scheduled, $n - 1$ execute 10 operations, while 1 executes 100× that. Figure 12a displays the overall throughput, while Figure 12b displays the average response time of the long-running transactions. We also use TiDB for comparison, as it provides both optimistic (`TiDB-O`) and pessimistic (`TiDB-P`) modes.

Without the priority mode enabled on the long-running client, a long-running transaction in TiQuE, just like with TiDB-optimistic, takes the entire duration of the benchmark to complete with just one regular client (Figure 12b), as the conflict probability is quite high. Conversely, TiQuE's priority mode causes the long-running transactions to abort, at most, one time, behaving similarly to the TiDB with locking. However, TiDB is faster, as although a transaction might need to wait, it never needs to be retried. Although improving long-running transactions, TiQuE's priority mode causes a decrease in the overall throughput (Figure 12a), as now short transactions are the ones being aborted. As for the native MonetDB, long-running transactions actually have the same success probability as regular ones in this workload, given the first writer acquires a column lock. This explains the irregular throughput and response time.

5 RELATED WORK

Although transactional isolation has traditionally been a core part of monolithic database management systems [5, 11, 12, 14, 18, 19], there are multiple proposals to implement it also as a self-contained layer on top of NoSQL or as part of layered NewSQL systems. A key driver of innovation is the ability to support transactions in mainly analytical or hybrid systems.

Transactional layers for NoSQL systems. Transactional layers for NoSQL stores extend the existing interface with transactional demarcation and use different external custom-built components. Some exploit the ability of the data store to hold multiple versions of data items. Omid [41, 56] builds on HBase which natively offers only single-row atomicity [4] and provides Snapshot Isolation using a separate centralized manager for transaction validation. HBaseSI [112] and the work by V. Padhye et al. [84] use HBase tables to order transactions on commit. Likewise, Megastore [38] adds transactions to Bigtable [44]. Others, such as CloudTPS [107] and pH1 [46], do not require multi-version storage by managing versions explicitly. In any case, none of the underlying data stores provide query execution capabilities and, unlike TiQuE, manipulation of transactional information in these systems is done by custom hard-coded logic which cannot easily be changed or tuned to particular workloads.

Distributed SQL and NewSQL. Distributed SQL and NewSQL systems require distributed transaction management instead of or in addition to a traditional implementation of transactions within the buffer management layer. They rely on distributed consensus [73, 83] for totally ordering transactions on commit for deterministic certification [3, 13, 17, 22, 35, 49, 66, 94, 94, 100, 104, 108], assume that different sites are responsible for updating different data and then use *two-phase commit* for atomic commitment [29, 43, 49, 95, 100, 108], or resort to locking the entire read/write-set [77, 102]. PolyphenyDB [105] extends transactional guarantees to polystores, i.e., systems comprised of different database management systems [97], but uses two-phase locking to handle conflicts and requires all locks at the start. Moreover, it uses table or partition-level locks [62]. In sharp contrast to all these proposals, TiQuE does not require in-depth changes to the engine implementation and our assumptions can also be met by distributed systems. Early work has shown that a similar approach is useful for polystores [54].

Transactions for analytical processing. There is a body of work targeting incremental updates with transactional guarantees to large datasets, kept, for example, in cloud storage providers. Percolator [86] provides reliable incremental updates to Bigtable by implementing Snapshot Isolation with a client library and a timestamp oracle. DeltaLakes [36] aims at providing incremental updates to Parquet [106] files. Unlike TiQuE, these systems are adequate only for very large granularity updates. The implementation strategy of TiQuE, of using tables to store multiple versions of data, also resembles the technique known as Slowly Changing Dimensions [70]. This is however aimed at storing historical data for queries over time. It does not do certification and thus does not provide general-purpose isolation.

HTAP on separate data. Hybrid processing faces the core challenge of reconciling a row-based or N-ary Storage Model (NSM) for transactional performance with a columnar or Decomposition Storage Model (DSM) for analytics [48]. A common approach is to support also columnar storage in traditional row-based systems [7, 74, 82, 90]. However, columnar tables often have fewer features (e.g., Postgres C-Store [82] does not support updates or deletes). Partition Attributes Across [30] (PAX) aims at the middle ground between NSM and DSM by storing all fields of a record on the same page but co-locating fields of the same column together.

HTAP by full replication. The problem can be circumvented by using separate OLTP and OLAP systems and replicating data among them using periodic *Extract Transform Load* (ETL) jobs [89]. As this comes with the cost of managing two separate systems and manually handling replication, some solutions manage this architecture as a whole. For instance, F1 Lightning [109], SAP HANA ATR [75], and Databus [50] capture data modifications on an OLTP database and asynchronously send them to OLAP replicas. TiDB [63] replicates data from OLTP to an OLAP format with asynchronous log replication in the Raft[83] protocol. BatchDB [80] is also similar, but only stores one version per row in the OLAP database. OctopusDB [52] takes this to the extreme and considers multiple copies of the data on several layouts, named storage views, and the query optimizer chooses the optimal one for a specific query. The main drawback with these systems is that they require full copies of data and the OLAP mirrors are often considerably staler in relation to their OLTP counterparts, affecting real-time analytics. Additionally, it increases the operational costs of managing multiple database systems. This is however a testament to the difficulty in modifying an existing storage engine for HTAP, that TiQuE addresses.

HTAP without full replication. Some proposals aim at dynamically managing partial copies in the same database system. A hybrid architecture for operational reporting [93] assumes that past data are more often read than updated, storing it in a DSM store, but also stores recent operational data there in row format, for fast inserts and real-time analytics. HyPer [67] and H^2Tap [34] use memory *copy-on-write* to maintain transient OLAP snapshots. Oracle In-Memory [72] keeps a copy of specific tables or table partitions in a columnar format in-memory, which must be first materialized either when the database starts or when the partition is first queried (configurable). SingleStore [87], L-Store [92], and NoisePage [78] store recently modified data in a row format and the remaining in OLAP-optimized columnar format. Other systems change data layout automatically based on the workload. Data Morphing [59] and HYRISE [58] automatically organize pages similarly to PAX. Peloton [37, 85] and H_2O [32] also move data between formats based on the past workload. TiQuE has some similarities with these systems, mainly in the management of data with different ages in different structures and the ability to optimize young data for transactional workloads, although without changing the storage engine.

Advanced transactional techniques. As executing large analytical queries concurrently with short, operational transactions in multi-versioned systems, independently of the storage model, increases the volume of versions that must be managed, solutions such as Diva [69], KVell+ [76], Steam[42], and SIRO[68] improve obsolete version purging and reduce the version lookup domain, thus reducing memory requirements and read complexity. Although our proof-of-concept relies on the *keep everything until the oldest transaction* approach for simplicity, it can also be adapted to support a finer-grained garbage collection. Finally, there has been substantial work to overcome some limitations imposed by the optimistic execution model: To mitigate the amount of work wasted when conflicts are frequent, solutions such as batching and reordering [51] and MRVs [53] reduce the abort probability; and solutions such as TicToc [111] and Silo [103] have also been proposed to avoid the impact of serial timestamp assignment on optimistic executions under extreme loads [110].

6 CONCLUSIONS AND FUTURE WORK

Our key hypothesis underlying TiQuE is that transactional operations such as reconstructing the snapshot and validating conflicts, in the context of a hybrid workload, are themselves challenging data processing operations that benefit from an optimizing query engine. Our results demonstrate this: First, we show that different queries and execution plans should be chosen for metadata manipulation for different workloads, in sharp contrast to the traditional approach of hard-coding them in the buffer management layer. This allows us, for instance, to easily provide a workaround for the impact of long-lived updated transactions. Second, we improve transactional performance in MonetDB precisely by exploiting the ability to use optimized execution plans for the management of transactional information itself. This allows us to obtain performance results in the same order of magnitude as state-of-the-art HTAP solutions with a fraction of the development effort.

These results open up interesting future possibilities: First, TiQuE should be applicable to a wider range of distributed data processing and NewSQL systems, that have limited or no transactional capabilities. This should provide a path to hybrid processing without the need for profound re-engineering to accommodate transactions. Second, the resulting flexibility makes it interesting and feasible to explore the possibility of supporting a wider range of isolation criteria, ranging from transactional causal consistency [31, 79], to even serializability [55], by themselves and in combination in the same system. On the other hand, this vision would benefit from improved meta-programming capabilities in database systems, such as views, rules, triggers, and in general, the ability to intercept and override significant events.

ACKNOWLEDGMENTS

Special thanks to the anonymous reviewers for their helpful feedback. This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

REFERENCES

[1] 2021. Snapshot Isolation in SQL Server. https://learn.microsoft.com/en-us/dotnet/framework/data/adonet/sql/snapshot-isolation-in-sql-server.

[2] 2022. Elle - Black-box transactional safety checker based on cycle detection. https://github.com/jepsen-io/elle.

[3] 2022. Galera Cluster. https://galeracluster.com/.

[4] 2022. Hive Transactions. https://hbase.apache.org/acid-semantics.html.

[5] 2022. IBM Db2 Database. https://www.ibm.com/products/db2-database.

[6] 2022. Jepsen - Distributed Systems Safety Research. https://jepsen.io.

[7] 2022. MariaDB ColumnStore. https://mariadb.com/kb/en/mariadb-columnstore/.

[8] 2022. MonetDB. https://www.monetdb.org/.

[9] 2022. MonetDB Documentation - Optimizer Pipelines. https://www.monetdb.org/documentation-Jan2022/admin-guide/performance-tips/optimizer-pipelines/.

[10] 2022. MonetDB Documentation - User Defined Functions. https://www.monetdb.org/documentation-Jan2022/dev-guide/sql-extensions/user-defined-functions/.

[11] 2022. MySQL. https://www.mysql.com/.

[12] 2022. Oracle Database. https://www.oracle.com/database/.

[13] 2022. Percona XtraDB Cluster. https://www.percona.com/software/mysql-database/percona-xtradb-cluster.

[14] 2022. PostgreSQL. https://www.postgresql.org/.

[15] 2022. PostgreSQL Documentation – 13.2.2. Repeatable Read Isolation Level. https://www.postgresql.org/docs/14/transaction-iso.html#XACT-REPEATABLE-READ.

[16] 2022. PostgreSQL Documentation – 15. Parallel Query. https://www.postgresql.org/docs/14/parallel-query.html.

[17] 2022. Riak Docs - Strong Consistency. https://docs.riak.com/riak/kv/latest/developing/app-guide/strong-consistency/index.html.

[18] 2022. SQL Server. https://www.microsoft.com/sql-server.

[19] 2022. SQLite. https://www.sqlite.org/.

[20] 2023. Apache HBase 2.2.3 API - Increment. https://hbase.apache.org/2.2/devapidocs/org/apache/hadoop/hbase/client/Increment.html.

[21] 2023. MongoDB Documentation - Transactions. https://www.mongodb.com/docs/v6.0/core/transactions/.

[22] 2023. MySQL 8.0 Reference Manual - Chapter 18.1.1.2 Group Replication. https://dev.mysql.com/doc/refman/8.0/en/group-replication-summary.html.

[23] 2023. MySQL Documentation - 15.7.2.3 Consistent Nonlocking Reads. https://dev.mysql.com/doc/refman/8.0/en/innodb-consistent-read.html.

[24] 2023. PingCAP Docs - TiDB Optimistic Transaction Model. https://docs.pingcap.com/tidb/v6.5/optimistic-transaction.

[25] 2023. PingCAP Docs - TiDB Transaction Isolation Levels. https://docs.pingcap.com/tidb/6.5/transaction-isolation-levels.

[26] 2023. SingleStore Docs - Durability. https://docs.singlestore.com/db/v8.0/en/introduction/faqs/durability.html.

[27] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[28] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No. 00CB37073)*. IEEE, 67–78.

[29] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2007. Sinfonia: a new paradigm for building scalable distributed systems. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 159–174.

[30] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance.. In *VLDB*, Vol. 1. 169–180.

[31] Deepthi Devaki Akkoorath, Alejandro Z Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 405–414. https://doi.org/10.1109/ICDCS.2016.98

[32] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. 2014. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 1103–1114.

[33] Peter Alvaro and Kyle Kingsbury. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (2020), 268–280. https://doi.org/10.5555/3430915.3442427

[34] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The case for heterogeneous HTAP. In *8th Biennial Conference on Innovative Data Systems Research*.

[35] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 163–172.

[36] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3411–3424.

[37] 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.

[38] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[39] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record* 24, 2 (1995), 1–10.

[40] Philip A Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading.

[41] Edward Bortnikov, Eshcar Hillel, Idit Keidar, Ivan Kelly, Matthieu Morel, Sameer Paranjpye, Francisco Perez-Sorrosal, and Ohad Shacham. 2017. Omid, Reloaded: Scalable and {Highly-Available} Transaction Processing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 167–180.

[42] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable garbage collection for in-memory MVCC systems. *Proceedings of the VLDB Endowment* 13, 2 (2019), 128–141.

[43] Prima Chairunnanda, Khuzaima Daudjee, and M Tamer Özsu. 2014. ConfluxDB: Multi-master replication for partitioned snapshot isolation databases. *Proceedings of the VLDB Endowment* 7, 11 (2014), 947–958.

[44] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 1–26.

[45] Edgar F Codd. 2002. A relational model of data for large shared data banks. In *Software pioneers*. Springer, 263–294.

[46] Fábio André Castanheira Luís Coelho, Francisco Miguel Barros da Cruz, Ricardo Manuel Pereira Vilaça, José Orlando Pereira, and Rui Carlos Mendes de Oliveira. 2014. pH1: a transactional middleware for NoSQL. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. IEEE, 115–124.

[47] 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.

[48] George P Copeland and Setrag N Khoshafian. 1985. A decomposition storage model. *Acm Sigmod Record* 14, 4 (1985), 268–279.

[49] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 1–22.

[50] Shirshanka Das, Chavdar Botev, Kapil Surlaker, Bhaskar Ghosh, Balaji Varadarajan, Sunil Nagaraj, David Zhang, Lei Gao, Jemiah Westerman, Phanindra Ganti, et al. 2012. All aboard the Databus! Linkedin's scalable consistent change data capture platform. In *Proceedings of the third ACM symposium on cloud computing*. 1–14.

[51] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control through Transaction Batching and Operation Reordering. *Proc. VLDB Endow.* 12, 2 (oct 2018), 169–182. https://doi.org/10.14778/3282495.3282502

[52] Jens Dittrich and Alekh Jindal. 2011. Towards a One Size Fits All Database Architecture.. In *CIDR*. Citeseer, 195–198.

[53] Nuno Faria and José Pereira. 2023. MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting. *Proc. ACM Manag. Data* 1, Article 43 (May 2023). https://doi.org/10.1145/3588723

[54] Nuno Faria, José Pereira, Ana Nunes Alonso, and Ricardo Vilaça. 2022. Towards Generic Fine-Grained Transaction Isolation in Polystores. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer International Publishing.

[55] Alan Fekete. 2018. Serializable Snapshot Isolation. In *Encyclopedia of Database Systems*, Ling Liu and M. Tamer Özsu (Eds.). Springer, 3476–3479. https://doi.org/10.1007/978-1-4614-8265-9_80774

[56] Daniel Gómez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, and Maysam Yabandeh. 2014. Omid: Lock-free transactional support for distributed data stores. In *2014 IEEE 30th Intl. Conf. on Data Engineering*. 676–687. https://doi.org/10.1109/ICDE.2014.6816691

[57] Jim Gray and Andreas Reuter. 1992. *Transaction processing: concepts and techniques*. Elsevier.

[58] 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.

[59] Richard A Hankins and Jignesh M Patel. 2003. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings 2003 VLDB Conference*. Elsevier, 417–428.

[60] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD Intl. Conf. on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[61] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Found. Trends Databases* 1, 2 (Feb. 2007), 141–259. https://doi.org/10.1561/1900000002

[62] Marc Hennemann and Marc Vogt. 2022. Polypheny-DB Repository - Pull request 408: Refactor Transaction Locking. https://github.com/polypheny/Polypheny-DB/pull/408.

[63] 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.

[64] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[65] S Idreos, F Groffen, N Nes, S Manegold, S Mullender, and M Kersten. 2012. MonetDB: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* (2012).

[66] Bettina Kemme and Gustavo Alonso. 2000. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *VLDB*. Citeseer, 134–143.

[67] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP OLAP main memory database system based on virtual memory snapshots. In *2011 IEEE 27th Intl. Conf. on Data Engineering*. 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[68] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived transactions made less harmful. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 495–510.

[69] 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.

[70] Ralph Kimball. 2008. Slowly Changing Dimensions, Part 2. https://www.kimballgroup.com/2008/09/slowly-changing-dimensions-part-2/.

[71] Hsiang-Tsung Kung and John T Robinson. 1981. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)* 6, 2 (1981), 213–226.

[72] 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.

[73] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), 51–58.

[74] 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.

[75] Juchang Lee, SeungHyun Moon, Kyu Hwan Kim, Deok Hoe Kim, Sang Kyun Cha, and Wook-Shin Han. 2017. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1598–1609.

[76] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 425–441.

[77] Justin Levandoski, David Lomet, and Kevin Keliang Zhao. 2011. Deuteronomy: Transaction support for cloud data. In *Conference on innovative data systems research (CIDR)*.

[78] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (dec 2020), 534–546. https://doi.org/10.14778/3436905.3436913

[79] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 401–416.

[80] 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.

[81] Chandrasekaran Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)* 17, 1 (1992), 94–162.

[82] Hadi Moshayedi. 2014. PostgreSQL Columnar Store for Analytic Workloads. https://www.citusdata.com/blog/2014/04/03/columnar-store-for-analytics/.

[83] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*. 305–319.

[84] Vinit Padhye and Anand Tripathi. 2013. Scalable transaction management with snapshot isolation for NoSQL data storage systems. *IEEE Transactions on Services Computing* 8, 1 (2013), 121–135.

[85] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[86] Daniel Peng and Frank Dabek. 2010. Large-scale incremental processing using distributed transactions and notifications. (2010).

[87] 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.

[88] F Raab. 1993. Overview of the TPC benchmark C: a complex OLTP benchmark. *Chapter* 3 (1993), 131–267.

[89] Ravishankar Ramamurthy, David J DeWitt, and Qi Su. 2003. A case for fractured mirrors. *The VLDB Journal* 12, 2 (2003), 89–101.

[90] 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.

[91] David Patrick Reed. 1978. *Naming and synchronization in a decentralized computer system.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[92] Mohammad Sadoghi, Souvik Bhattacherjee, Bishwaranjan Bhattacharjee, and Mustafa Canim. 2018. L-Store: A Real-time OLTP and OLAP System. https://doi.org/10.5441/002/edbt.2018.65

[93] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. 2008. A hybrid row-column OLTP database architecture for operational reporting. In *International Workshop on Business Intelligence for the Real-Time Enterprise*. Springer, 61–74.

[94] Dharma Shukla. 2018. Azure Cosmos DB: Pushing the frontier of globally distributed databases. https://azure.microsoft.com/pt-pt/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/.

[95] Yair Sovran, Russell Power, Marcos K Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 385–400.

[96] Michael Stonebraker. 1987. The Design of the POSTGRES Storage System. In *Proceedings of the 13th Intl. Conf. on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 289–300.

[97] Michael Stonebraker. 2015. The Case for Polystores. ACM SIGMOD Blog. (2015). https://wp.sigmod.org/?p=1629

[98] Michael Stonebraker and Uĝur Çetintemel. 2018. " One size fits all" an idea whose time has come and gone. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 441–462.

[99] Hironobu Suzuki. 2021. The Internals of PostgreSQL: Chapter 5 Concurrency Control. Retrieved 2022-10-14 from https://www.interdb.jp/pg/pgsql05.html

[100] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.

[101] Dixin Tang, Hao Jiang, and Aaron J. Elmore. 2017. Adaptive Concurrency Control: Despite the Looking Glass, One Concurrency Control Does Not Fit All. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p63-tang-cidr17.pdf

[102] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[103] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32.

[104] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1041–1052.

[105] Marco Vogt, Nils Hansen, Jan Schönholz, David Lengweiler, Isabel Geissmann, Sebastian Philipp, Alexander Stiemer, and Heiko" Schuldt. 2021. Polypheny-DB: Towards Bridging the Gap Between Polystores and HTAP Systems. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, Vijay "Gadepally, Timothy Mattson, Michael Stonebraker, Tim Kraska, Fusheng Wang, Gang Luo, Jun Kong, and Alevtina Dubovitskaya (Eds.). Springer International Publishing, Cham, 25–36.

[106] Deepak Vohra. 2016. Apache parquet. In *Practical Hadoop Ecosystem*. Springer, 325–335.

[107] Zhou Wei, Guillaume Pierre, and Chi-Hung Chi. 2011. CloudTPS: Scalable transactions for Web applications in the cloud. *IEEE Transactions on Services Computing* 5, 4 (2011), 525–539.

[108] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-latency transaction processing for globally-distributed data. In *Proceedings of the 2018 International Conference on Management of Data*. 231–243.

[109] 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.

[110] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (nov 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[111] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 Intl. Conf. on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935

[112] Chen Zhang and Hans De Sterck. 2011. HBaseSI: Multi-row distributed transactions with global strong snapshot isolation on clouds. *Scalable Computing: Practice and Experience* 12, 2 (2011), 209–226.