MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting

NUNO FARIA, INESCTEC and University of Minho, Portugal
JOSÉ PEREIRA, INESCTEC and University of Minho, Portugal

Performance of transactional systems is degraded by update hotspots as conflicts lead to waiting and wasted work. This is particularly challenging in emerging large-scale database systems, as latency increases the probability of conflicts, state-of-the-art lock-based mitigations are not available, and most alternatives provide only weak consistency and cannot enforce lower bound invariants. We address this challenge with Multi-Record Values (MRVs), a technique that can be layered on existing database systems and that uses randomization to split and access numeric values in multiple records such that the probability of conflict can be made arbitrarily small. The only coordination needed is the underlying transactional system, meaning it retains existing isolation guarantees. The proposal is tested on five different systems ranging from DBx1000 (scale-up) to MySQL GR and a cloud-native NewSQL system (scale-out). The experiments explore design and configuration trade-offs and, with the TPC-C and STAMP Vacation benchmarks, demonstrate improved throughput and reduced abort rates when compared to alternatives.

CCS Concepts: • **Information systems → Data management systems**; **Parallel and distributed DBMSs**; **Distributed database transactions**; **Database transaction processing**.

Additional Key Words and Phrases: concurrency control, transactions, distributed databases

ACM Reference Format:
Nuno Faria and José Pereira. 2023. MRVs: Enforcing Numeric Invariants in Parallel Updates to Hotspots with Randomized Splitting. *Proc. ACM Manag. Data* 1, 1, Article 43 (May 2023), 27 pages. https://doi.org/10.1145/3588723

1 INTRODUCTION

High throughput transactional systems exploit parallelism in processor cores and distributed nodes with multiple concurrent operations. Transactional isolation does however mean that invalid outcomes (anomalies) are avoided and that a correctness criterion such as *Serializability* or *Snapshot Isolation* is met [7]. This creates the illusion that each transaction is executing alone, greatly simplifying application development.

Unfortunately, transactional workloads often exhibit hotspots [39, 45, 63]: Some items are accessed by concurrent transactions with high probability, meaning that locking and validation mechanisms used for isolation have a severe impact on usable throughput. Hotspots arise in stock trading, shopping, banking, and numerous applications. Some are as simple as counting events, such as user votes or advertisement impressions in Web sites. Some of these applications, such as prepaid telco plans, selling event tickets, or keeping track of remaining inventory, in addition to

Authors' addresses: Nuno Faria, nuno.f.faria@inesctec.pt, INESCTEC and University of Minho, Portugal; José Pereira, jop@di.uminho.pt, INESCTEC and University of Minho, Portugal.

2836-6573/2023/5-ART43 $15.00
https://doi.org/10.1145/3588723

counting, also need to enforce a bound invariant, that ensures that the quantity being tracked does not cross a set threshold.

Besides considering their impact in high-performance concurrency control systems [14, 21, 61, 70], dealing with hotspots in numerical values by explicitly allowing parallelism has attracted considerable interest, and a spectrum of proposals exists. For traditional database systems based on locking, *escrow locking* greatly reduces contention by avoiding long-lived locks [38]. In main memory many-core database systems, *phase reconciliation* splits the quantity into multiple variables that can be independently accessed by different cores [36]. Both solutions are forms of reservations [5, 42], in which parts of the total value are set aside for different operations.

In this paper, we address the same issue in emerging distributed or cloud-based database management systems such as Spanner [11], Aurora [62], CockroachDB [58], SingleStore [44], MySQL GR [12], and MongoDB Replica Set [25]. These systems are challenging as *escrow locking* is not applicable (e.g., locks in MySQL GR are local and transactions in different nodes run optimistically), distributed synchronization has a considerable impact on latency, or the unpredictability makes them ineffective (e.g., how many separate splits are needed in a serverless system). Although there are other reservation mechanisms, they focus on mitigating round-trip times and availability during partitions and do not solve contention or are unable to enforce a lower bound. Section 6 discusses previous proposals and explains their shortcomings in this new environment. Moreover, we address the challenge of providing a solution that works in current cloud-based and off-the-shelf systems, that is, using only their application programming interfaces.

In a nutshell, our proposal is a form of *value splitting*, as we split a contended value into n parts and allow concurrent transactions to access each of them. In systems where a row is atomic in terms of isolation (i.e., row-level locking or validation), this means splitting a contended value over multiple records, hence the name of Multi-Record Values (MRVs). As described in Section 2.2, our main insight is using randomness to route each client operation: Each access starts from a random position and efficiently traverses an index to one or more records as needed, without static assignment or knowing the number of splits. Our second insight is that existing transactional isolation provides all the coordination needed between concurrent clients and maintenance tasks, thus avoiding additional explicit distributed interaction. Finally, we dynamically balance and maintain the appropriate number of records for each item fully in the background, as made possible by the random access technique. Therefore, our main contributions are:

- The proposal of the Multi-Record Values technique, including operations and maintenance algorithms (Section 2);
- A thorough discussion of how the proposal is implemented, including a middleware-level implementation that can be used in existing applications (Section 3);
- An evaluation of the design decisions and how system parameters influence performance (Section 4);
- An evaluation with different benchmarks and database management system architectures, also in comparison with state-of-the-art alternatives [14, 36, 38, 61, 70] (Section 5).

All the code, scripts, and raw data is available online.[1]

2 MULTI-RECORD VALUES

The proposal for Multi-Record Values relies on how the value in a particular column is split into multiple records in an auxiliary table; on intercepting and modifying the operations for adding and subtracting to that value; and on background worker tasks that adjust and balance the values stored in multiple records.

[1] https://github.com/nuno-faria/mrv

2.1 Assumptions

We assume a database structured as collections of data items. This can be a relational database, where each collection is a table and each item a row with multiple columns, or it can also be an extended relational or document model, with a nesting structure. To simplify the presentation, we refer to them as *tables* and *rows*, although we later describe implementations in document-oriented systems such as MongoDB.

We also assume that there is an index providing efficient random access to rows by a composite key. Although not strictly necessary, we also take advantage of sequential traversal by key order. This maps to tree-structured indexes found in most database systems [6, 37]. When a composite key is not available, it can be simulated by concatenating the columns in a way that respects ordering [15].

We assume that a simple equi-join operation traversing a 1-to-n relation can be done, either by a query engine or directly by the application by looking up the relevant rows. We can also take advantage of *views*, which make this transparent to the application by exposing a logical table that encapsulates the join operation, and of *rules* that translate updates to such logical table to the original physical tables. Note that logical views are not materialized, do not add storage overhead, and do not cause consistency issues.

We also assume multi-operation transaction isolation and recovery with fine-grained row-level locking or validation. MRVs preserve the underlying isolation level and are compatible with optimistic (OCC) and pessimistic (*two-phase locking*, or 2PL) systems. MRVs are not compatible with *Serializability* resulting from table-level locking, but work on row-grained *Serializability*, such as *Serializable Snapshot Isolation* [10, 41]. The key requirement is that read and update operations on different rows of the same table do not conflict. We also assume that transactions can be rolled back on request, as a result of a failed validation, or to break deadlocks, recovering according to the selected isolation level.

2.2 Key insights

The general strategy in Multi-Record Values (MRVs) is to change the database schema by replacing the column holding the contended numeric values with a separate table that keeps multiple records for each original row, using the original table's key and a unique record identifier. To add or subtract to the value, one needs to add or subtract to any of these records. To read the current value, one needs to sum them all. Given the assumptions of fine-grained transaction isolation, this preserves application semantics and correctness while avoiding update conflicts. Moreover, the read operation with *Snapshot Isolation* does not cause conflicts and is still very efficient. The key insight and contribution of MRVs is how each transaction is assigned to a physical record and how the various records, holding parts of the total value, are managed efficiently.

First, as different clients might have different access patterns, we avoid statically assigning them to records (as done to processor cores in *phase reconciliation* [36] or nodes in most distributed reservation systems [5]). To ensure that accesses are evenly spread, our first insight is to use a random number, for each access, to determine which record to use. Assuming that the number n_i of records for each item i is big enough, this results in a small probability of conflict. This avoids the need for explicit coordination of clients, which would be costly in a distributed environment.

Subtractions might not be fully possible on a single record if its current value is lower than what is being subtracted. Thus, the subtraction might require multiple accesses to complete. This could be done simply by keeping the remainder after a first subtraction and carrying it on to a second random one, and so forth, until it is fully done. This, however, makes it difficult to determine unsuccessful termination, which happens after all records have been visited and there is still a

remainder. Moreover, when the underlying mechanism is using exclusive locks on update, accessing multiple items in a random order increases deadlock probability. We address this by performing only one random lookup and then scanning to the next record. After the last record ($n_i - 1$) we wrap around and restart the process on the first (0), treating the records for each item as a circular structure. As we will show in Section 2.4, variations of this algorithm also work for efficiently filtering with a lower bound without full materialization, by stopping when the target is met.

Second, we address the issue of reconfiguring the number of records for each item i such that it remains optimal in face of changing workloads. This means that n_i is no longer a constant and needs to be determined for each item before generating the random number to access a record. Storing n_i for each item means additional space overhead and might be an additional source of contention. One alternative is to determine n_i before each access by counting the number of records. However, this would introduce overhead that is larger precisely for those items that are accessed more frequently and thus need more records, and potentially an extra round-trip from the application to the server. In fact, the mere act of counting the rows can conflict with any concurrent updates.

Our second insight is to decouple the random number used for lookup from the number used for the records' keys as follows: We select some constant N such that $N \gg n_i$ for all items i; We tag each record with a random number in $[0, N[$; For each access, we generate a random rk' between $[0, N[$ and choose the record with the smallest key $\geq rk'$, or the smallest of all if none exists. Interestingly, with a tree-structured index, the lookup and scan operations are the same, with the same cost as those with sequentially labeled records. The remaining challenge is how to dynamically adjust the number of records for each item and balance existing value among them, while minimizing the space and time overheads.

2.3 Data structure

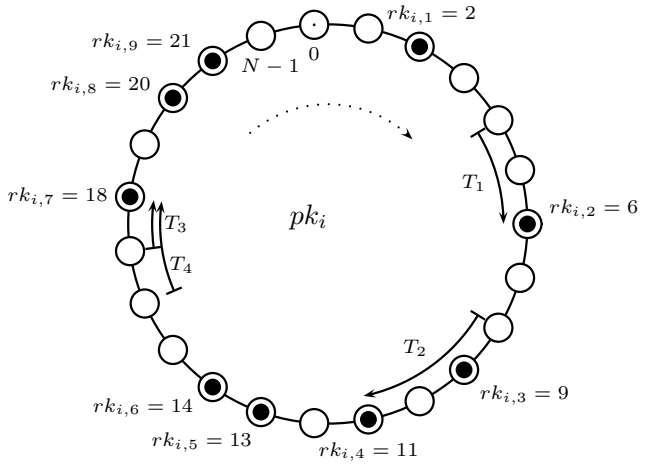
A column v in some table with primary-key pk is transformed into a Multi-Record Value (MRV) as follows: An additional table is created with 1-to-n relation from the original one, established by the original's primary key (pk_i). We have thus one or more reservation records with $v_{i,j}$ amount for each original value v_i. Each of these is identified by the pair $(pk_i, rk_{i,j})$, where $rk_{i,j}$ is a unique integer between $[0, N[$. Therefore, the original value v_i corresponding to pk_i is no longer stored in the original table but reconstructed by joining both tables on pk_i and then as the sum of $v_{i,j}$, for all j. In database management systems without a relational join operation, this means a second lookup to the values table.

This data structure means that parts of a value – records with the same pk_i – are seen as organized in a ring structure of size N. Figure 1 represents this logical ring for some item i with primary key pk_i and currently holding some value v_i. Solid black circles represent currently existing records, holding reservations of the value, while empty circumferences represent indices not yet assigned to any record. Note that the current number of records is not kept anywhere or used in any of the operations and that a separate logical ring exists for each item (i.e., row) in the original table.

Besides the core advantage of splitting a value for concurrent access, this structure has the interesting property that multiple accesses by the same transaction are ordered by rk, which reduces the possibility of introducing deadlocks on accesses to the various records for the same value. The downside is that the original row is split across two tables, which impacts memory locality and processor cache usage. This means that MRVs should be judiciously applied only to columns containing concurrency hotspots.

2.4 Operations

We now describe the following operations: *read* (read the current value), *write* (write a new value), *bound* (check the value for a lower bound), *add* (add to the value), and *sub* (subtract from the value



(pk	rk)	v
	...	
pk_i	2	5
pk_i	6	3
pk_i	9	2
pk_i	11	4
pk_i	13	4
pk_i	14	6
pk_i	18	5
pk_i	20	1
pk_i	21	4
	...	

Fig. 1. Diagram of the structure of MRV pk_i with $n_i = 9$, the respective table representation, and various possible transactions alternatives ($T_1..T_4$).

with an implicit bound). The key advantage of MRVs is that *add*, *sub*, and *bound* can be combined or executed concurrently without conflicts, when possible, but still enforcing transactional isolation. With *Snapshot Isolation* (SI), *reads* are conflict-free, leading to some performance advantages but needing a slightly different algorithm for the *bound* operation as follows.

To read (*read*) some value v_i, one scans all corresponding records and sums $v_{i,j}$. This conflicts with all other operations that might be updating the same value (or none if using SI or similar).

To write (*write*) some value v_i, one deletes all existing records and inserts a new one with a random rk' in the range $[0, N[$. This conflicts with all other operations that might be accessing the same value (or, with SI, only with those updating the same value).

To add (*add*) some value to v_i, one must first pick a random rk' in the range $[0, N[$ to lookup a record. The record selected is the one whose rk is equal to rk' or, if it does not currently exist, the one that comes immediately after rk'. This is shown in Figure 1 by transaction T_1, which starts with $rk' = 4$, which does not exist, and goes on to use $rk = 6$. In an ordered index such as a B-tree, the operation to select the first existing rk given some random rk' results in a simple index scan (unless it has to wrap around).

To subtract (*sub*) some δ from v_i while enforcing that $v_i \geq 0$, one starts with a random rk' and completes if the next $v_{i,j} \geq \delta$. If not, $v_{i,j}$ is set to 0 and the remainder carries on to the next record. Figure 1 illustrates this with T_2, that starts with $rk' = 8$, sets $v_{i,3}$ to 0, and subtracts the remainder from $v_{i,4}$. Again, a key advantage of this strategy is that a second lookup can be avoided by continuing the index scan to the next value. Finally, an attempt to subtract more than the total value will iterate over the entire MRV and, once it reaches back to the initial record, it will still not be done. In this case, it will have to rollback to preserve the lower limit of zero.

To check a lower bound (*bound*), one proceeds to visit enough records $v_{i,j}$ to meet the desired quantity. We can also combine *bound* with a *sub* that decrements a smaller value (e.g., decrement by 1 if greater than 10). Note however that with SI, one needs to update each visited record with its current value to ensure that concurrent *sub* operations do not invalidate them.

In Figure 1, note that T_1 and T_2 do not conflict and can be adding, subtracting, and checking the invariant at the same time, possibly in different sites in a distributed system. The probability of conflict can be decreased by increasing the number of records, but conflicts cannot be fully avoided.

For instance, in Figure 1, T_3 starts with $rk' = 16$, T_4 starts with $rk' = 17$, and both end up trying to modify the value of $rk_{i,7} = 18$. MRVs are designed to take advantage of the underlying concurrency control mechanisms. This means that one of T_3 or T_4 is automatically rolled back/locked by the database system, as both attempt to update the same physical record.

Note also that these operations rely on the underlying system providing multi-operation transactional isolation and recovery. For instance, a *sub* relies on triggering a conflict with operations that are concurrently subtracting from the same records (blocking/aborting one of them) and being able to undo partial updates. An interesting corollary is that multiple operations on the same or on different values can be combined in a single transaction.

2.5 Background worker tasks

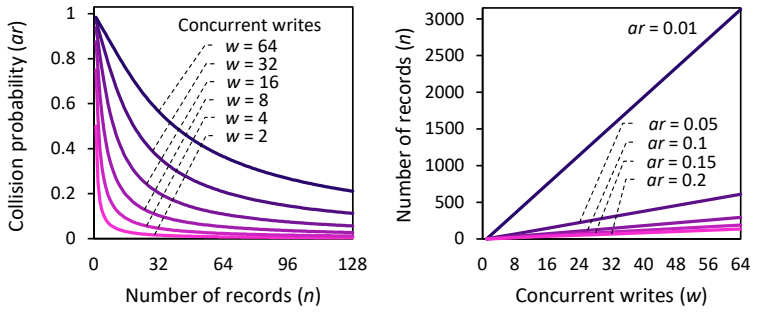
A set of background worker tasks determines the appropriate n_i for each item and, if $n_i > 1$, splits values among the resulting n_i records, to minimize both conflicts and overhead. Adjusting the number of records or rebalancing values needs to be done dynamically to react to changes that create new hotspots. Thus, the base for the background workers is the ability to keep a directory of items that are currently update hotspots and estimate how many conflicts are caused by each of them. The key insight here is, precisely because we are addressing hotspots, that relevant items are a small minority of all stored items and state can be kept in a rolling window. This makes MRVs efficient even in databases with a huge number of values, that can be adjusted and balanced with minimal impact on foreground workload. Analogous to operations on MRVs, both workers also rely on the underlying isolation for correctness.

The *adjust* worker determines how many records are needed for each item. Increasing the number of records decreases conflict probability (Section 2.6) but harms read performance and storage size, thus we cannot blindly set n_i to an arbitrarily high value. In addition, zero-value records are counterproductive for subtractions.

This worker is implemented by defining an acceptable conflict rate – ar_{goal} – and proportionally changing the number of records such that the observed conflict rate converges to that. We also consider an absolute minimum (nr_{min}) and maximum (nr_{max}) number of records for each MRV, useful with bursty workloads and extreme contention. A *min_avg_value_per_record* can also be set to avoid growing the number of records when the total value nears zero, as conflicts are then inevitable. Finally, we also consider an ar_{min}, below which we consider removing records as the lower abort rate does not justify the higher read and storage overheads. Although these targets might vary with applications, we found them to be stable, and for all the evaluation results, $ar_{goal} = 5\%$ and $ar_{min} = 1\%$.

The *balance* worker redistributes the total value v_i among the n_i existing records for each item. This is needed as common workloads may lead to an imbalance in the distribution of the total value among records and degrade performance. For example, a stock of some product would probably have frequent *sub* operations of a few units – i.e., clients buying the product – and less frequent *add* operations of many units – i.e., the store restocking. This would lead to most records having a zero value and some having large amounts, thus defeating the purpose of having multiple ones.

Perfect balancing is achieved by reading the value and equally dividing the amount (*all* algorithm). Although this ensures that records end up balanced in one iteration, it also means that it has to update the entire set, which conflicts with any concurrent update and thus will likely require a few tries to succeed. We found out that the best alternative for database systems that do not acquire locks on reading is to select the K-maximum and K-minimum records based on their amount and balance the partial sum among them. We call this the *minmax* algorithm. It greatly improves convergence when dealing with a relatively high number of records. For systems that acquire



(a) ar based on n (b) n needed to reach some ar

Fig. 2. Relation between collision probability (ar), number of records (n), and number concurrent writes (w) in MRVs.

shared locks for reading, finding the max and min might lead to contention. In this case, we can randomly select k items and balance the partial sum among them (*random* algorithm).

The *balance* worker is further optimized by executing only if the difference in percentage between the maximum and minimum selected records is greater or equal to some value (defaults to 10%). Additionally, in situations where there are a large number of subtractions and just a few large additions, the *add* operation can itself be distributed over a small subset of low-value records.

To provide the information on current hotspots needed by both workers we consider another structure – *Tx_Status*, volatile and updated asynchronously – to count commits and concurrency-induced rollbacks to MRVs. If a transaction updates MRVs and succeeds, the respective commit counters are incremented. If a transaction aborts while updating an MRV, the respective abort counter is incremented. Periodically, *Tx_Status* is cleaned as only recent information is relevant.

2.6 Collision and complexity analysis

Intuitively, increasing the number of records in an MRV (n) reduces collision probability, while increasing the number of concurrent writes (w) has the opposite effect. To determine the collision probability (ar), MRV writes can be modeled as a generalization of the Birthday Problem – where n represents the days and w represents the people – assuming each write only accesses one record, which should be the norm. As only one concurrent write per record will commit, ar is simply determined by w minus the records accessed, divided by w. To find the number of records accessed, we first compute the number of records that have not been written. The probability of some record having no accesses is $(1 - \frac{1}{n})^w$ (the first write not picking it, $\times$ the second write not picking, and so on), thereby the number of records without any write is n times that. Consequently, the number of records accessed is n minus the records without any writes. Thus, the collision probability becomes:

$$ar(n, w) = \frac{w - (n - n \cdot (1 - \frac{1}{n})^w)}{w} \tag{1}$$

Figure 2a shows how ar evolves based on n and w. Increasing w will increase the collision probability for the same n. It is also clear from Figure 2a that as the target ar tends to zero, we will need an exponentially higher n. Figure 2b exhibits this in detail, where the n required to reach some ar, based on w, increases exponentially with the decreasing ar target. On the other hand, for the same ar target, n increases linearly with the increasing contention.

Increasing n, although reducing the collision probability, will negatively impact the time complexity of the MRVs operations. Assuming a tree-structured index over the MRV, the lookup operation

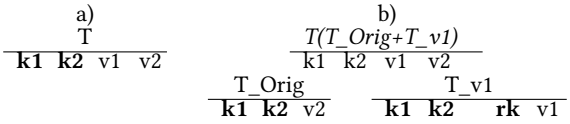


Fig. 3. Before (a)) and after (b)) converting column $v1$ of table T into MRV. T_Orig is the original table without $v1$; T_v1 contains the records of MRV column $v1$; and T is now a view that reconstructs the original table by joining T_Orig with T_v1.

has a complexity of $O(\log(n))$. This means that the *add* operation has a complexity of $O(\log(n))$, as only one lookup is required. Meanwhile, the *sub* and *bound* operations have a complexity of $O(n)$, as one lookup is needed and then possibly the traversal of all records. However, with balanced records, a single iteration will suffice and, in practice, will be as efficient as *add*. Finally, the *read* and $write$ operations have $O(n)$ complexity. Nevertheless, if an MRV is not a hotspot, $n = 1$, making all these operations $O(1)$ by default.

3 IMPLEMENTATION

We implement multiple proof-of-concepts of Multi-Record Values, baselines, and competing alternatives. Besides being used in experiments in Sections 4 and 5, these implementations illustrate how MRVs can be approached with three different strategies; how they can be applied in data systems with very different architectures and interfaces; and finally, how they suit various applications.

3.1 Strategies

MRVs admit an efficient application-level implementation that makes it possible to judiciously address hotspots without changes to the database management system, being ideal for closed-source cloud services. It is used for the microbenchmark, described in Section 3.2, in PostgreSQL, MySQL GR, MongoDB, and a closed-source cloud-native system (*System X*).

For each column to be transformed into MRV, we move it to an additional table (or collection). We then modify queries that read the current value to sum all records. We use additional threads in the application and explicitly collect commits and concurrency-induced rollbacks for background worker tasks. These data are then inserted asynchronously in the Tx_Status table (or collection).

In this strategy, both reads and writes must be translated by the application, either by manually changing the SQL code or with a wrapper that performs that translation. If the system supports views, as is the case with PostgreSQL, MySQL, *System X*, and MongoDB, then the application does not need to translate reads. Instead, we create a regular view in place of the original table to perform the computation. If the system also supports rules, as is the case with PostgreSQL [20], then writes can also be handled by the database engine, without changes to the application. We call this the middleware-level strategy, using it with PostgreSQL for the more complex benchmarks, namely, TPC-C and STAMP Vacation.

The middleware implementation can be used as follows: First, we specify the columns to model as MRVs and the configuration parameters. A Python script will then do the following for each table T containing MRVs: The table is renamed to T_orig, keeping only non-MRV columns; a table is created for each MRV column *mrv_name* that stores the modeled MRVs (T_mrv_name); and a view with the original table's name, which joins the tables and computes MRVs' $total\_value$, is also created. Figure 3 provides an example of a table where we want to model $v1$ as MRV. Next, the Insert/Update/Delete statements over T are rewritten using rules. Finally, background worker tasks are implemented using a standalone daemon program. We identify each MRV in *Tx_Status* with the corresponding table name and primary key.

Table 1. SQL API for application and middleware-level MRVs. All operations target MRV $\texttt{MRV}_j$ of table $\texttt{T}_i$. `gt: MRV` $> v$, `gte: MRV` $\geq v$.

Operation	SQL Code (`SELECT ...`)	Middleware
$Read(k)$	`sum(v) FROM` $\texttt{T}_i\texttt{\_MRV}_j$ `WHERE pk=`k	View
$Add(k, \delta)$	`update_`$\texttt{T}_i\texttt{\_MRV}_j(k, \delta)$	Update rule
$Sub(k, \delta)$	`update_`$\texttt{T}_i\texttt{\_MRV}_j(k, -\delta)$	Update rule
$Bound(k, v)$	`(gt|gte)_`$\texttt{T}_i\texttt{\_MRV}_j(k, v)$	-
$Write(k, w)$	`write_`$\texttt{T}_i\texttt{\_MRV}_j(k, w)$	-
$Insert(row)$	`insert_`$\texttt{T}_i(row)$	Insert rule
$Delete(k)$	`delete_`$\texttt{T}_i(k)$	Delete rule

Table 1 specifies the SQL API for applicational and middleware-level MRVs. In the middleware implementation, the *bound* operation cannot be translated and the Write operation is not fully transparent due to the ambiguity with *add*/*sub*.

Finally, implementing MRVs at engine-level can use a similar approach, but fully hiding the auxiliary structures from the client. This results in greater transparency and can allow DDL (Data Definition Language) statements to be used by the application. Another option is to model the additional tables and metadata as internal structures and offer MRVs as a new data type. In this option, the workers can be fully implemented at engine-level and MRVs can achieve minimum overhead. We use it in DBx1000 [69], which implements an experimental in-memory data system targeted at high-concurrency workloads. Here, MRV records are stored using array structures and thus lookups are performed using position indices.

3.2 Benchmarks

To find the optimal parameters and evaluate MRVs' performance, we consider the following benchmarks:

Microbenchmark – Models the stock of several products, used to explore the effective overhead/performance gain in a wide range of configurations. Operations can subtract or add several units to a random product (write) or compute a product's total stock (read). Unless otherwise stated, each test runs for 65 seconds, with the first 5 removed from the results. Compiled with `openjdk-11`.

TPC-C – TPC-C [46] is used as an example of how MRVs perform in a more general, widely used workload. The top conflicts, presented in Table 2, led us to model add-only columns *w_ytd* and *d_ytd* as MRVs. Column *d_next_o_id* is not modeled as MRVs to not violate its uniqueness property, as required by TPC-C.[2] Each test runs for 60 seconds. Executed with `sysbench 1.0.20`[40].

TPC-C DBx1000 – Another TPC-C implementation to compare against *escrow locking* and different concurrency techniques (DBx1000[68]). It executes only the *payment* transaction – as it is the only one where MRVs and *escrow locking* are used – modified to evaluate add and subtract performance. No workers are used due to the additional implementation complexity. Each test runs until 100k transactions are committed. Compiled with `g++-7`.

STAMP Vacation – The Vacation benchmark [35] contains a large number of increment, decrement, and read operations on numeric values. The topmost abort causes, presented in Table 3, led us to model columns *numFree* and *numTotal* of the different items as MRVs. Each test runs for 60 seconds. Compiled with `openjdk-11`.

[2]Conflicts generated from monotonically increasing identifiers, such as *d_next_o_id* in TPC-C, can be avoided with auto-incremented fields, with the caveat of being non-contiguous if a transaction aborts after advancing them, or with non-monotonic identifiers (e.g., UUIDs). Otherwise, there is no parallel solution to this problem [18].

Table 2. Top 3 most common abort causes in the TPC-C benchmark (Repeatable Read). The presented queries concern *updating a warehouse's year-to-date by some amount* (1), *incrementing a district's next order identifier* (2), and *updating a district's year-to-date by some amount* (3).

Statement	%
`UPDATE Warehouse SET w_ytd += __h_amount ...`	37
`UPDATE District SET d_next_o_id += + 1 ...`	28
`UPDATE District SET d_ytd += __h_amount ...`	27

Table 3. Top 3 most common abort causes in the STAMP Vacation benchmark. The presented queries concern *decrementing a reservation's stock* (1,2) and *incrementing a reservation's stock* (3).

Statement	%
`UPDATE car_reservation SET numFree -= 1 WHERE id = $1`	52
`UPDATE flight_reservation SET numFree -= 1 WHERE id =$1`	28
`UPDATE room_reservation SET numFree += 1 WHERE id = $1`	14

3.3 Database management systems

We consider five different database management systems:

PostgreSQL – PostgreSQL 12 represents a traditional centralized SQL system, also relevant as a managed cloud service. It is an example of how MRVs can be implemented transparently to the application. We use Repeatable Read – which results in *Snapshot Isolation* – and Read Committed isolations – which does not generate aborts on concurrent updates but serializes them with statement locking [19].

MongoDB – We use MongoDB 4.2.2 as our NoSQL single-writer cluster, deployed in a *replica set* to support transactions [25]. To have a consistency similar to *Snapshot Isolation*, we use the following setup: `Read preference`: *primary*, to avoid stale data aborts [24]; `Read concern`: *snapshot* [23]; `Write concern`: *majority* [26].

MySQL Group Replication – MySQL Group Replication is a multi-writer architecture where transactions execute optimistically and, at commit, are totally ordered, using consensus, and then certified and committed/aborted by all sites [12]. MySQL Server 8.0.17 is used in three sites with Repeatable Read isolation.

System X – *System X* is a codename for a geo-distributed, multi-writer, closed-source NewSQL database management system. Since ensuring geo-availability incurs a higher commit latency due to synchronous replication, it is a perfect target for MRVs. Additionally, it is an example of a system where we cannot modify the internal engine but still can make use of MRVs with an application-level implementation. Read-write transactions execute under Serializable guarantees.

DBx1000 – DBx1000 [69] is used for the MRVs engine-level implementation and to compare with *escrow locking* and high-performance concurrency control. It uses Serializable isolation.

4 PARAMETERS AND DESIGN DECISIONS

This section explores the impact of configuration parameters and design decisions. Besides the impact of the number of records for each item, we consider dynamically adjusting the number of records to the workload, balancing the values, and using a limited history window for the workers.

For this purpose, we use the microbenchmark of Section 3.2. All tests are executed on Google Cloud Engine virtual machines (N1 Series), configured with Ubuntu 18.04 LTS. The tests with

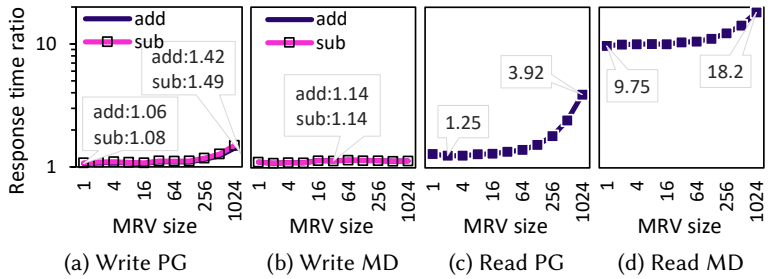


Fig. 4. Comparison of the response time ratio between MRVs and baseline (single record) in the microbenchmark, for PostgreSQL (PG) and MongoDB (MD).

PostgreSQL use a 24 vCPUs, 24 GB RAM, and 500 GB SSD instance. The tests with MongoDB use a Replica Set for transactional isolation with 3 instances of 8 vCPUs, 8 GB RAM, and 165 GB SSD.

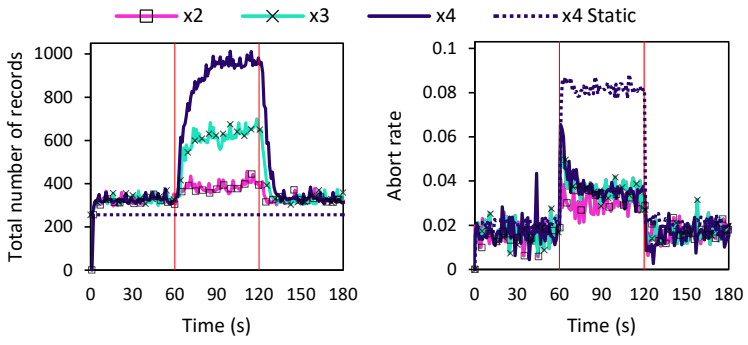
4.1 Number of records

We assess the impact of varying the number of records for each item in the read and write operations by measuring the average response times. The *adjust* worker is disabled in these experiments, such that the number of records is constant. Each test executes with 1 client and 1 product (meaning there are no conflicts), performing either a *read* (read test) or an *add*/*sub* of multiple units (write test). In the MRV write, the *sub* operations should only need to update one record, since frequent adds and the balance worker keep the records with enough stock. Figure 4 presents the write results, using PostgreSQL (Figure 4a) and MongoDB (Figure 4b), and the read results, using PostgreSQL (Figure 4c) and MongoDB (Figure 4d).

Write results for PostgreSQL (Figure 4a) show that the *add* response time is 1.06× (1 record) to 1.42× (1024 records) higher than the baseline (i.e., before splitting the target value to a separate MRVs table), while the *sub* response time is 1.08× to 1.49× higher. The write response time when dealing with a high number of records is the result of a more expensive index lookup. As for the MongoDB results (Figure 4b) the ratio is on average 1.12× higher. The lower overhead is explained by the native operation itself having a higher response time than in PostgreSQL (3ms vs 1ms), so the overhead is not as noticeable. These are encouraging results, as they present a low overhead for a low number of records while the higher overhead for a higher number of records can be easily justified by reducing the collision probability on hotspots.

Read results with PostgreSQL (Figure 4c) show a higher difference when comparing to the writes, being between 1.25× and 3.92×. Although this confirms that the MRV read complexity grows with the number of records, it doesn't increase linearly with its number (e.g., between 1 and 1024 records, the response time only increases by a factor of 3). MongoDB results (Figure 4d) also follow a similar evolution. However, the ratios are considerably higher, being between 9.75× and 18.18×. The main cause is that we rely on the aggregation framework, which is more expensive than the find instruction used by the baseline code.

Note that the baseline read results are relatively small, e.g., with PostgreSQL being 0.08ms, which makes the MRVs with 1024 records (0.31ms) seem comparatively high. However, the baseline write is 12× more expensive than the baseline read, meaning that even with an equal number of writes and reads, tackling write performance can be easily justified by the more expensive reads. Section 5 confirms that the reduction in the conflict probability in workloads with hotspots offsets the overhead of reads. Also note that as MRVs are dynamic, only hotspots will suffer higher overheads, while solutions relying on static splitting must pre-allocate in excess.



(a) Number of records (b) Abort rate

Fig. 5. Evolution of the number of records and abort rate based on different load changes in MRVs. $\times n$ means an n times increase in the number of clients between 60 and 120 seconds.

Table 4. MRVs storage overhead relatively to the baseline.

# of clients	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**
1 hotspot	*1*	*43*	*119*	*182*	*393*	*683*	*1k*	*1k*	*1k*	*1k*
1 column	2.03	2.25	2.43	2.25	2.29	2.25	2.18	2.25	2.09	2.55
TPC-C	1.00	1.00	1.01	1.02	1.05	1.08	1.09	1.01	1.01	1.00
1 col. static	~250									

4.2 Adjust worker

Next, we show how the adjust worker makes MRVs adapt to changing workloads. To do so, we consider the following test: We start with 8 clients and 256 products, which results in a small probability of conflict; At the 60-second mark, we increase the number of clients by 2×, 3×, and 4×; Finally, at the 120-second mark, we decrease the number of clients back to the original 8. We also apply the 4× test to MRVs with no *adjust* worker (*Static*), to serve as a baseline. Figure 5a displays the total number of records over time, while Figure 5b displays the overall abort rate over time.

The dynamic results show that in the first 60 seconds, the total number of records is relatively stable with 1 record per product (Figure 5a). With the increased load at the 60-second mark, there is an abort rate spike in all tests, above the established goal of 0.05 (Figure 5b). However, the adjust worker adds more records, depending on the abort rate seen for each product. As the load increases, so does the probability of conflict, meaning different loads will require different numbers of records to meet the goal. As the load decreases back to the original at the 120-second mark, the adjust worker starts removing unnecessary records, which in practice means lower read and storage overheads. On the other hand, static splitting shows that while the initial records are enough for the initial load, they do not adapt when it increases, raising the abort rate. The alternative would be to pre-allocate in excess, trading off lower abort rates for higher execution and storage overheads.

Reducing conflict probability with MRVs increases the amount of storage space used. The results in Table 4 display the storage used in relation to the native single-record baseline, based on the number of clients. The first (*1 hotspot*) shows the impact on a single hotspot value. As expected, it increases with contention and converges to the nr_{max} configuration parameter. This does not, however, reflect the cost of MRVs, as only a few values turn out to be hotspots. The second (*1 column*) shows the space overhead on a single column modeled as MRVs with 100k records and

client accesses according to a *power law* distribution, that creates realistic hotspots. In this case, the overhead ranges from 2 to 2.55. Finally, as real data is likely to contain many columns with only a few modeled as MRVs, we check the space overhead in the TPC-C benchmark with one warehouse (high contention). In this case, the space overhead is negligible with at most 9% over the baseline. This should be the norm for the average practical use case. Table 4 also shows that the storage overhead of pre-allocating enough records in static splitting to meet the target abort rate in the *1 column* test is around 250× higher than the baseline, or more than 100× higher comparatively to MRVs, again exemplifying the limitations of static splitting.

In short, the adjust worker is able to adapt to a changing workload and configure the number of records necessary to reduce the probability of conflict. Moreover, in realistic applications with a skewed workload leading to some hotspots in only a few columns, the storage overhead is negligible.

4.3 Balance worker

Next, we consider the optimal algorithm to balance values among existing records, in particular, trying to avoid records with zero value that are useless but add to the overhead. The first tests, presented in Figure 6, compare the execution time (Figure 6a), percentage of records with zero value (Figure 6b), throughput (Figure 6c), and abort rate (Figure 6d) of the different algorithms, with no balancing background worker as a baseline (*none*). The benchmark runs under the *uneven* operation distribution (on average, one 1000-unit add for every 1000 1-unit subs), as generating more unbalanced records is the worst-case scenario for the worker. It uses 64 clients, 64 products (same access probability) with no initial stock, to simulate a low-stock workload, and a variable number of initial records per product (no *adjust* worker). The *balance* worker runs every 100ms. Both *random* and *minmax* balance two records every iteration.

We first conclude that, with a low number of records per product, all three algorithms perform in a similar fashion, which is expected since at a low number both *minmax* and *random* will tend to equal *all*. In addition, we can already notice the benefit of using a balance worker, given that without one it results in twice as many zero-valued records (Figure 6b). However, as the number of records increases, the differences become evident. *random* will tend to perform the same as the alternative with no worker, since the probability of selecting two zero-value records for balancing increases, wasting an iteration (Figure 6b). The *minmax* and *all* algorithms perform similarly in terms of throughput up to 32 records per product (Figure 6c). After that, *all* loses a considerable amount of performance, as forcing an update to all records in one transaction results in a higher abort rate (Figure 6d) and long iteration times (Figure 6a). Meanwhile, *minmax* not only presents a higher throughput (Figure 6c) and lower abort rate than the other alternatives (Figure 6d), it also iterates as fast as *random* (Figure 6a). The exception is after 128 records, as the worker does not balance two records that have the same value, which is often the case with *random*. Thus, *minmax* will be used for all remaining tests.

We consider next how *minmax* behaves based on the number of records it updates in one iteration. A K of 1 means it will balance the max and min records (default), a K of 2 means it will balance the two max and two min records, and so on. A K of $\frac{\#\text{records}}{2}$ equals the *all* algorithm. Since $K = 1$ provides the optimal execution time and $K = \frac{\#\text{records}}{2}$ achieves equilibrium in just one iteration, there should be some K that is the optimal trade-off. A similar experiment to the one in Figure 6 is performed, presented in Figure 7. Each cell represents the product of the number of zero-valued records and balance time ratios in relation to $K = 1$, i.e., $c_i = \frac{\text{Zeros}_{K=i}}{\text{Zeros}_{K=1}} \cdot \frac{\text{Time}_{K=i}}{\text{Time}_{K=1}}$. As we want to minimize both, the optimal K is the one with the smallest c. Intuitively, if $K = 2$ results in half the number of zeros but doubles the execution time, $c = 1.0$ and so we consider it the same as $K = 1$. If $K = 2$ results in half the number of zeros but the same time, $c = 0.5$ and thus better than $K = 1$.

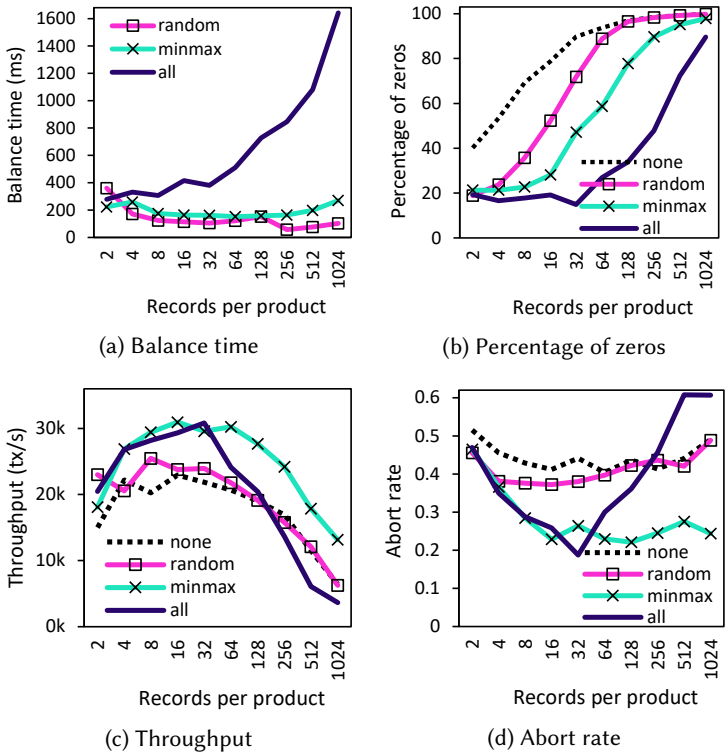


Fig. 6. Comparison between different balance algorithms.

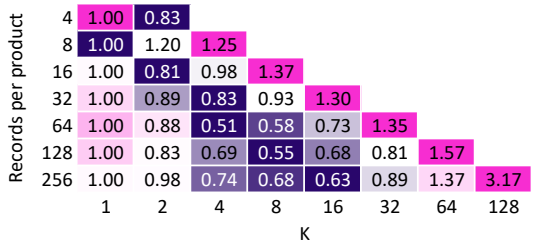
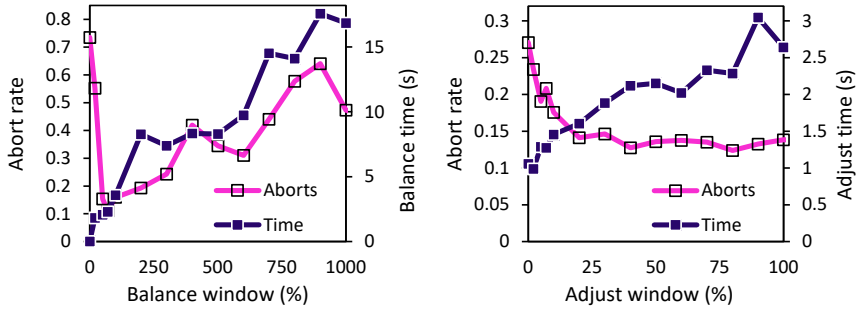


Fig. 7. Comparison between different balance sizes (K) in MRVs performance ($c_i = \frac{\text{Zeros}_{K=i}}{\text{Zeros}_{K=1}} \cdot \frac{\text{Time}_{K=i}}{\text{Time}_{K=1}}$). Lower is better.

By analyzing the results (Figure 7), we conclude that there is indeed a better K than 1 for most tests. A rough estimate let us conclude that $K_{opt} = 2$ when $\#records = 4$, $K_{opt} = \frac{1}{8} \cdot \#records$ when $\#records < 64$ and $K_{opt} = \frac{1}{16} \cdot \#records$ when $\#records \geq 64$.

4.4 Window size

Finally, we evaluate the impact of different *balance* and *adjust* windows – the oldest information that is considered for the balance/adjust workers. This is key to bounding the amount of memory used and is desirably not longer than the period of the worker. The microbenchmark's *accessDistribution* is set to *power law*, meaning a product's access probability is $(x + 1)^{-1}$, where x is its index



(a) Balance worker (b) Adjust worker

Fig. 8. Effects of different windows in the MRVs workers.

($x \in [0, 99999]$). This models a workload with a large number of products but few hotspots. The balance tests (Figure 8a) execute only *subs*, with each product having a fixed 64 records, where one has the entire stock while the other 63 have none, to evaluate the balance effectiveness. The adjust tests (Figure 8b) execute *adds* and *subs* evenly and each product starts with one record. The window is defined as a percentage of the worker's respective period that is for balance and adjust workers, respectively, 100ms and 1000ms.

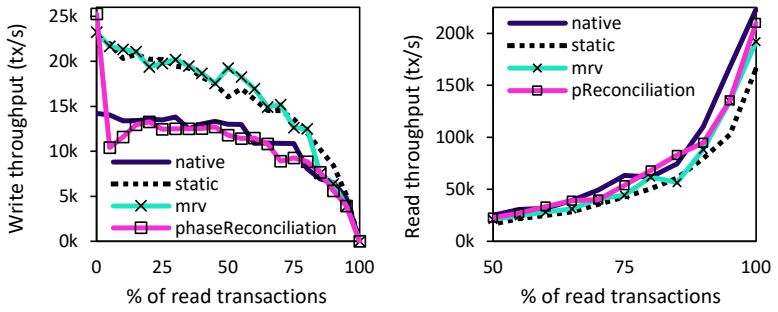
The balance tests (Figure 8a) show that a window of 0%, which is equivalent to not using the worker, results in a client abort rate of around 70%, clearly undesirable. Increasing the window until 100% quickly decreases the abort rate, with a relatively low impact on the execution time. However, growing the window past that point results in a considerably higher execution time, as the worker must consider a larger set of products, and in turn, actually outputs a higher abort rate. The optimal window for this workload is between 50% and 100%. The adjust tests present a similar trend. A window of 0% results in the highest abort rate, while increasing it up to 40% results in a substantially lower one. After that, the adjust time increases while the abort rate stays roughly the same. We consider the optimal adjust window for this workload to be 25%. Although different workloads might have different optimums, we use these values for the remaining tests and achieve good results.

5 PERFORMANCE EVALUATION

This section compares MRVs with competing approaches and then to a native numeric column as a baseline in different environments, including centralized and distributed systems. We use the same experimental conditions of Section 4 for PostgreSQL and MongoDB. Likewise, tests with DBx1000 use a 24 vCPUs, 24 GB RAM instance. The tests with MySQL Group Replication run on 3 instances of 8 vCPUs, 8 GB RAM, and 165 GB SSD. And finally, *System X* uses 3 read-write replicas deployed in relatively close proximity to the 24 vCPU client.

5.1 Comparison with Phase Reconciliation

First, we compare MRVs to *phase reconciliation* [36], which is also a form of reservations and close to MRVs in terms of the problem addressed. In contrast to other reservation mechanisms, instead of partitioning by site, *phase reconciliation* partitions by core in its *split* phase. Additions and some subtractions can execute concurrently, while reads must wait for the *joined* phase, where the partial values are merged into a single global one (phase is specific for each record).



(a) Write throughput (b) Read throughput

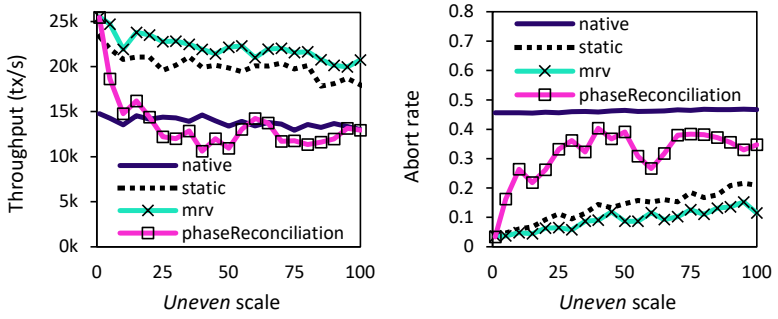
Fig. 9. Comparison between baseline (native), MRV, and *phase reconciliation* using the microbenchmark with a variable read percentage (PostgreSQL Repeatable Read).

To implement the *phase reconciliation* technique we use a split for each client. The coordinator determines if a record changes phase every 20ms, as specified by the paper. The conditions that trigger a phase change are, however, not as clear, as exact values were not specified in the paper. We thus consider three different metrics: abort rate per record, clients waiting for the *joined* phase, and transactions that aborted due to no stock. We optimized the first two metrics for the first workload using a grid search algorithm, where we found that it is best to convert a record to the *split* phase when the abort rate per record is $> 65\%$, and to convert a record to the *joined* phase when 1) there is a client waiting for that record and 2) the total ratio of clients waiting for the *joined* phase is $> 25\%$. Finally, we also optimized the third metric, this time for the second workload, and found that it is best to join a record as soon as there is a client which hit a no-stock abort for that record.

We compare *phase reconciliation* with MRVs using the microbenchmark with two workloads (PostgreSQL Repeatable Read, 32 clients, 32 products): The first uses a variable ratio of read/write transactions, which allows us to see how both techniques adapt to varying workloads[3] (Figure 9; we show the read results from 50% to 100% for readability, as they are similar under 50%); The second uses the *uneven* operation distribution (just writes) with varying scales (1 to 100), where "1" means one *add* of value 1 for every *sub*, while "100" means *add* 100 for every 100 *sub* operations of value 1, in addition to populating each record with a reduced initial stock of 1000 (Figure 10). Many *subs* and few *adds* is the worst-case scenario for both solutions, as will it lead to more out-of-stock records. Thus, it will let us better compare the MRVs balance worker with the *phase reconciliation*'s join and split to distribute stock among records, as well as measure the impact of static splitting for reduced stocks. We also show static splitting (32 records; good baseline for these tests) to emphasize the relevance of background workers on performance, specifically the *adjust* in the first test and the *balance* in the second.

The first conclusion from the results with a variable percentage of read transactions (Figure 9) is that, with only writes or only reads, *phase reconciliation* is slightly better than MRVs, with 9% better throughput for both cases. For writes, *phase reconciliation* completely removes conflicts, while MRVs only reduce their probability. For the reads, the MRVs SQL code has a higher planning cost at the engine, even though both alternatives use only one record for this extreme. However, running reads and writes concurrently show us the main drawback of *phase reconciliation*: having to choose between fast writes and blocking reads or slow writes and non-blocking reads. This

[3]Since this is a variable workload, the number of initial records per product in MRVs is set to just one, instead of the usual one record per client.



(a) Throughput (b) Abort rate

Fig. 10. Comparison between baseline (native), MRV, and *phase reconciliation* using the microbenchmark with various *uneven* scales (writes only; PostgreSQL R. Read). $x = 1$: 1-unit *add* per 1-unit *sub*; $x = 5$: 5-unit *add* per five 1-unit *subs*, and so on.

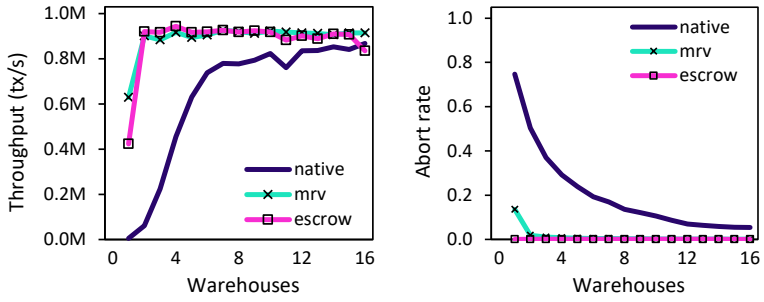
causes the overall performance to be similar to or even worse than the baseline. In fairness to *phase reconciliation*, we did not batch the reads and writes, which should result in higher throughput, as this would be more complex to implement. However, even with batching, it would still result in higher response times and, consequently, lower throughput than MRVs, as the transaction rate is dictated by the clients. MRVs, on the other hand, allow both reads and writes to execute concurrently, resulting in an overall higher throughput even if the both on their own are slightly slower than *phase reconciliation*. Regarding static splitting, although writes are faster than *phase reconciliation* and even match MRVs, the negative impact on reads is evident, given the over-allocation of records. Overall, in this test, static splitting is 14% slower than MRVs when reading.

Results with uneven numbers of adds and subs (Figure 10) show that increasing the *uneven* scale results in a considerable performance drop for *phase reconciliation*. As the stock takes longer to replenish and only affects one record at a time, the probability for the amount in some record to deplete increases, which will require changes to the *joined* phase, increasing overhead and conflict probability. On the other hand, by allowing subs to access multiple records, as well as employing the comparatively lighter balance worker, MRVs suffer only a small drop in performance as the skew between adds and subs increases. Without the *balance* worker, static splitting relies only on the increasingly sparse *adds* to keep records with stock, leading to a higher number of zero-valued records and increasing the number of conflicts for *subs* (on average, it has 50% more aborts than MRVs). However, static splitting is still faster than *phase reconciliation* in this workload, highlighting the performance impact of phase switching in dynamic workloads.

5.2 Comparison with Escrow Locking

Next, we compare MRVs with *escrow locking* [38], which is the proposal aimed at the same problem as MRVs in a centralized lock-based environment, using DBx1000 (with 2PL) [69]. *Escrow locking* is implemented by modifying the locking code to not acquire exclusive locks on *add* and *sub* operations, but instead manipulate an escrow local to each transaction within a row latch [38]. As for MRVs, we use 128 records for each item. Figures 11a and 11b show the throughput and abort rate, respectively (32 clients).

The results show that with one warehouse, which causes maximum contention, MRVs result in almost 50% higher throughput than *escrow locking*, but at the same time displays a higher abort rate, given the probabilistic nature of MRVs. As the number of warehouses increases, decreasing contention, the two options converge to similar results, meaning that the MRVs technique has



(a) Average throughput (b) Average abort rate

Fig. 11. Comparison between baseline (native), MRV, and *escrow locking* using TPC-C's *payment* in DBx1000.

comparable performance to *escrow locking* in a centralized database. This is good news, given that *escrow locking* is the closest proposal in terms of goals but restricted to a centralized lock-based implementation.

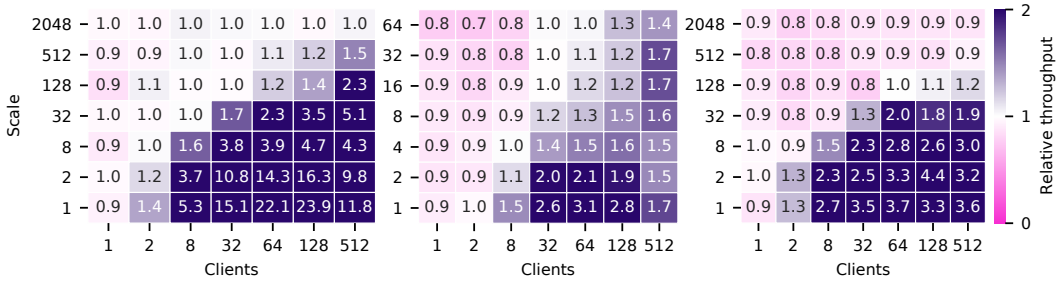
5.3 Varying workloads

We now compare the performance of MRVs to a native numeric column with different workloads: microbenchmark (Figure 12a), TPC-C (Figure 12b), and STAMP Vacation (Figure 12c). The initial number of records for the MRVs columns is the same as the number of clients, except for the Read Committed tests, where it is twice as much as there is no *adjust* worker, and the STAMP Vacation, which is based on a ratio between the number of clients and the number of items, to limit read overhead for low collision tests. We vary the number of clients and benchmark scale to evaluate how MRVs adapt in different scenarios. Results are presented as heatmaps that depict the throughput ratio between MRVs and native.[4] The bottom right corner of each chart displays situations of extreme contention, where dozens to hundreds of clients compete for write access to the same logic row, while the top left corner has little to no collisions.

Microbenchmark results (Figure 12a) show an increased throughput over the native as the number of clients grows and the number of products decreases, i.e., higher conflict probability. MRVs throughput ends up being up to 24× higher than with native numeric columns, as a result of a higher success probability. The only exception is the results with 1 client, where it ends up reaching around 0.9× the rate of the native since there are no conflicts, and as such MRVs offer nothing but overhead. Meanwhile, MRVs abort rate ranges between 1× and 0.01× (avg. 0.22×) higher than the native, while the response time ranges between 0.95× and 24× (avg. 2.36×), as in the native only the fastest transactions commit. Likewise, experiments with Read Committed result in similar improvements, even though no aborts are generated due to locking. It reaches up to 18× the native throughput, since a transaction in the single-record solution has longer wait times due to lock contention. Consequently, the response time is lower, ranging between 1.22× and 0.08× (avg. 0.76×). We also evaluated the microbenchmark with the *uneven* distribution (100 *subs* for a single *add*), where it reached up to 12× the native throughput, and with each transaction updating three products instead of one, reaching up to 27×.

TPC-C results (Figure 12b) show the MRVs overhead when the number of clients is low and the number of warehouses is high, i.e., a low conflict probability. This makes MRVs have around 0.9× the throughput of the native. As the number of clients increases, so does the throughput ratio, which reaches up to 3× more. Again, this is due to the reduced abort rate, being between 0.29×

[4]The corresponding charts for the abort rate and response time, can be found together with the source code.



(a) *Microbenchmark* (scale=Products)

(b) *TPC-C* (scale=Warehouses)

(c) *STAMP Vacation (opt.)* (scale=Items per type)

Fig. 12. Throughput comparison between MRVs and baseline (native) using different workloads with PostgreSQL's Repeatable Read. A value of 1.0 means MRVs and native have the same throughput, 2.0 means double the throughput for MRV, and so on.

and 1× (avg. 0.5×), while the response time is between 0.33× and 1.34× (avg. 0.88×). Although not shown, experiments with Read Committed also show improvements, reaching up to 2.6× the native throughput as a result of the lower response time (up to 0.36× higher). This shows that MRVs not only can be integrated with a more complex application, but also improve a workload's overall performance even when numerical write hotspots are only a small part of it.

Initial tests with the STAMP Vacation benchmark produced surprising results: Measured throughput varied widely in the baseline and, with an increasing workload, MRVs resulted in up to 40× higher throughput. Upon careful examination, we found that most problems were due to deadlocks, as operations randomly choose a set of services to acquire for one trip. By reducing conflicts, MRVs would thus also reduce deadlocks across different products. In addition, a small number of items led to stock depletion very quickly.

Thus, Figure 12c's results were obtained using the following modifications that reduce the disadvantage of native numeric columns and would likely have been implemented by application developers: Item writes are ordered by their ids, to avoid deadlocks; and the initial stock and the initial number of clients are increased. Furthermore, we also consider two optimizations for MRVs: we replaced the `numFree > x` predicates with our *bound* operation that does exactly that but more efficiently, in order to avoid, if possible, reading the entire set; and replaced the `UPDATES` to MRVs by manual calls to functions that do the same. The reason for this is because PostgreSQL's `UPDATE` rules always materialize the current value, resulting in a read per update.

The results show a positive correlation between a high number of clients/low rows per table and a high throughput ratio, where MRVs reach up to more than 4× the native's throughput, while the abort rate is between 0.09× and 1.18× (avg. 0.57×). Even though this benchmark is modeled using a higher number of numerical updates comparatively to the TPC-C, it uses even more reads and inserts,[5] explaining why it does not reach as high as the microbenchmark. Just like with TPC-C, we have some overhead for situations with low collision probability, where MRVs reach between 0.8× and 0.9× the native's throughput. Without the optimizations described above, MRVs reach up to 3.3× the native, while the overhead is slightly higher (between 0.7× and 0.9×). If we instead use static splitting (based on optimized MRVs), the overhead is considerably higher, reaching down

[5]In the most frequent procedure (*Make Reservation*), there are ten MRV reads, one regular read, four inserts, and just three MRV writes.

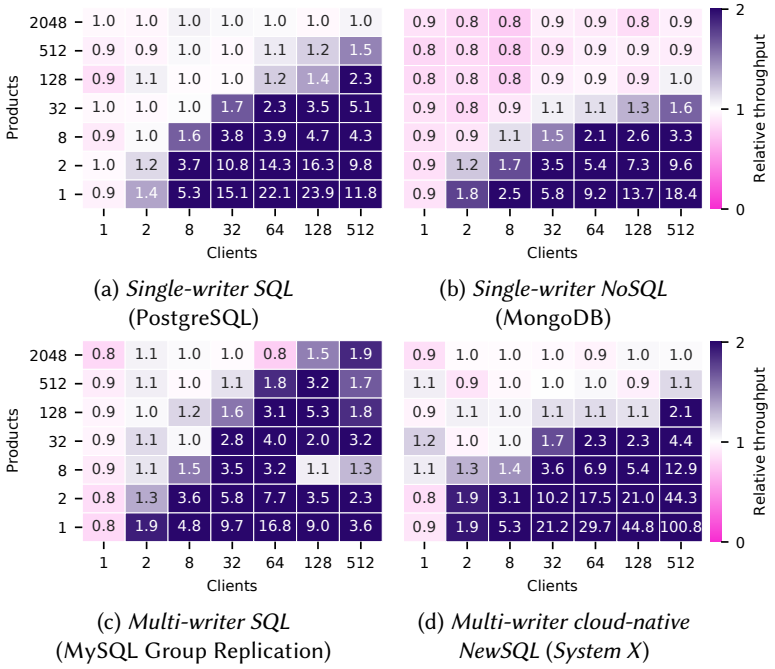


Fig. 13. Throughput comparison between MRVs and baseline (native) using the microbenchmark with different database management systems. A value of 1.0 means MRVs and native had the same throughput, 2.0 means double the throughput for MRV, and so on.

to 0.5× the native due to over-partitioning, while at the same time not reaching ratios as high as MRVs, as more records have depleted stock. Overall, MRVs are 15% faster than static ones.

In short, these tests show that MRVs have performance advantages whenever there are conflicts, even if in the context of varied multi-operation transactions and with the generic middleware. Moreover, they show a surprising advantage in reducing deadlocks and that minor changes to application code, that avoid the overhead of the generic implementation, are feasible and useful.

5.4 Varying database management systems

These experiments aim at evaluating MRVs on different database management systems, including distributed systems where solutions such as *escrow locking* are not applicable. To do so, the MRVs technique is compared to the native (single-record) versions in the microbenchmark using: a single-writer SQL system with PostgreSQL's `Repeatable Read` (Figure 13a); a single-writer NoSQL data store with MongoDB (Figure 13b); a multi-writer SQL database with MySQL Group Replication (Figure 13c); and a cloud-native, multi-writer NewSQL system with *System X* (Figure 13d). For *System X*, we model MRV rows with the key and rk encoded in a single column (e.g., `'p0.0123'`), as the composite key $\langle pk, rk \rangle$ led to false conflicts due to how the underlying storage works. Again, we rely on heatmaps showing the difference between MRVs and native, where the top left corner refers to low-collision runs while the bottom right corner refers to high-collision ones.

Figure 13 shows a pattern throughout all systems: Increasing the collision probability leads to relatively higher throughput with MRVs. We can also see that the MRVs in all SQL systems have a similar overhead, presenting a throughput of around 0.9× the native when there are no collisions. On the other hand, the NoSQL MRVs results present a relatively higher performance loss, even

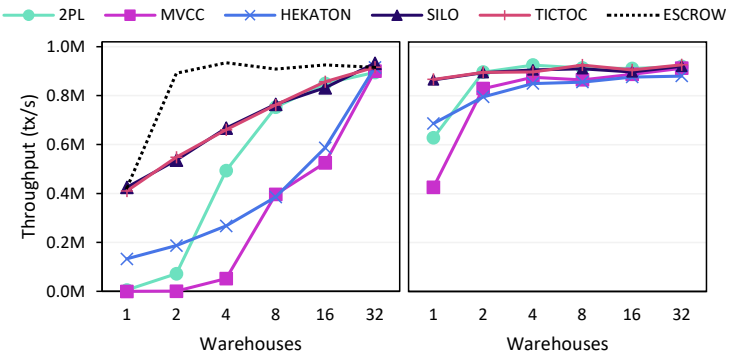


Fig. 14. Throughput comparison between different concurrency control techniques with and without MRVs. 2PL a) is equivalent to *native* of Figure 11; 2PL b) is equivalent to *mrv* in Figure 11.

when the number of clients is greater than one, as we can infer from the results with lower collision probability (128-2048), which is in line with the overhead presented in Figure 4b and is due to the limitations of its query processing mechanism.

Experiments with single-writer SQL systems show up to 24× the throughput of the corresponding native baseline, the single-writer NoSQL up to 18×, the multi-writer SQL up to 17×, and the multi-writer cloud-native NewSQL up to 100×, due to the lower abort rates (down to 0.01×, 0.01×, 0×, and 0× higher than the native, respectively), even if increasing response times (up to 24×, 1.36×, 1.7×, and 1.52×, respectively). Experiments with *System X* highlight one of the advantages of application-level MRVs: Improving the performance of update hotspots even when we do not have access to system internals. Furthermore, reducing conflict probability in cloud databases such as *System X* is especially important, as they are paid by the hour and by the maximum number of queries per second. This means that aborts not only have an impact on the response time seen by the client, but also limit the available resources and increase costs to meet demand.

In short, these experiments show that the MRVs technique is widely feasible and advantageous in a spectrum of different database management systems, including distributed and NoSQL systems.

5.5 Varying concurrency controls

One of the main advantages of MRVs is the fact that they can be layered on top of existing concurrency control, easing implementation and taking advantage of state-of-the-art optimizations. Figure 14 compares the throughput of various concurrency control techniques provided by DBx1000, using the same configuration as Figure 11 (TPC-C *payment*, 32 clients), with and without MRVs: common *two-phase locking* (2PL) [8] implemented with `WAIT_DIE` and *timestamp-ordering* implemented with *multi-version concurrency control* (MVCC) [48]; and high-performance Hekaton [14], Tictoc [70], and Silo [61]. *Escrow locking* is also evaluated [38].

The first conclusion is that MRVs, even when based on common 2PL and MVCC techniques, match or even outperform high-performance concurrency control techniques in high contention scenarios (i.e., very few warehouses). MRVs (Figure 14b) achieve the optimum with around 2-4 warehouses, for all techniques, while the MRV-less techniques (Figure 14a) require 32 warehouses. The second is that MRVs take advantage of better-performing underlying concurrency control techniques, as the combination with TicToc or Silo achieves close to optimum performance even with extreme contention.

Table 5. Comparison between MRVs and related work.

Technique	Limit invariant?	Primary target	Parallelism technique	Consistency level
Multi-Record Values (MRVs)	*yes*	*numeric fields*	*reservations + commutativity*	*strong*
Single record (baseline)	*yes*	-	-	*strong*
Escrow locking [38]	*yes*	*numeric fields*	*reservations*	*strong*
Phase reconciliation [36]	*yes*	*numeric and set fields*	*reservations + commutativity*	*strong*
RedBlue [28] / $\mathcal{I}$-confluence [3]	*yes*	*commutative operations*	*commutativity*	*strong/strong eventual*
Timestamp splitting [22]	*yes*	*records*	*multi-ts records*	*strong*
Parallel transaction chopping [17, 53, 64]	*yes*	*transactions*	*transaction chopping*	*strong*
Distr. reservations [4, 29, 33, 42, 66, 67]	*yes*	*inter-site numeric fields*	*distributed reservation*	*strong eventual*
Cassandra's Counters [65]	*no*	*inter-site numeric fields*	*distributed partitioning*	*strong eventual*
Delta transactions [57]	*no*	*numeric fields*	*commutativity*	*strong*
Counting sets [56]	*no*	*set fields*	*commutativity*	*strong eventual*
Operation transformation [16]	*no*	*various operations*	*delayed resolution*	*strong eventual*
CRDTs (w/o BCounter) [27, 51, 52]	*no*	*various fields*	*commutativity*	*strong eventual*
Post-Commit Rules [59, 60]	*no*	*various operations*	*delayed resolution*	*eventual*

6 RELATED WORK

Table 5 summarizes the main competitors to MRVs discussed in this section. We start off by comparing proposals addressing the same problem, i.e., update conflicts in numerical values and enforcing a lower bound, that are thus direct competitors to Multi-Record Values. A classical solution for locking systems is *escrow locking* [38]. Instead of acquiring a lock for the duration of a transaction, the change to the value is kept in escrow until transaction commit and, otherwise, undone. This is efficient when implemented as part of locking mechanisms in a database engine. The challenge with the distributed systems that we consider is that they get their performance from executing transactions locally and then validating writes at the end, in a single distributed step. In both cases, we cannot assign splits to individual nodes, either due to full replication (e.g, MySQL GR [12] and CockroachDB [58]) or layering (e.g., Spanner [11] and Aurora [62]), so a node does not own any rows exclusively. Managing an Escrow would need additional distributed coordination steps, for instance, to a central coordinator, which is precisely what these systems avoid for good performance. Although it has been proposed at the application-level and in a distributed system [54], given the overhead of remote synchronization, it is feasible only for long-lived transactions to cope with disconnection and not for fine-grained concurrency hotspots, unlike MRVs.

In memory-based systems, *phase reconciliation* [36] splits a record for each core when it becomes contended, by moving it from the *joined* (single record) to the *split* phase. As each core has its own partial value, this precludes conflicts. However, read operations must block until the record

is *joined* back. Additionally, for numeric values, subtractions with lower bound invariant must abort/block until the *joined* phase if there is not enough amount in the corresponding split. As shown in Section 5, explicit allocation of splits and the synchronization on phase change impose a significant overhead in systems other than memory-based multi-core databases.

Similarly for distributed systems, selecting the appropriate parallelism for each operation, either implicitly (e.g., RedBlue [28]) or explicitly from invariants (e.g., $\mathcal{I}$-confluence [3]), allows executing some operations asynchronously and in parallel in various sites, but is ineffective for update hotspots when enforcing a bound, as all subtract operations would have to be considered not parallel.

Second, there are high-performance concurrency control techniques [14, 21, 61, 70] that alleviate contention in general, mainly on validating the transaction, but are not specific to numerical records. *Timestamp splitting* [22] uses different timestamps for subsets of columns in the same table, allowing operation in different sub-sets to execute concurrently. Although our solution was designed to alleviate contention to updates of the same field, it also somewhat addresses the same problem as *timestamp splitting*, since it does not trigger conflicts with other MRVs or non-MRVs columns. *Timestamp splitting* also employs a technique to delay updates to commit time, which precludes conflicts but does not decrease response time, as updates are still serialized. It is also possible to improve performance by executing operations of the same transaction in parallel, in high-performance concurrency control based on the concept of *transaction chopping*, i.e., dividing a large transaction into multiple sub-transactions [17, 53, 64]. All of these are useful but do not address the issue of concurrent updates on the same numerical field. However, they are complementary to MRVs and can be easily and advantageously combined with them as shown in Section 5.

A third category includes reservation techniques for distributed systems [42, 43] in which different sites are assigned different parts of some value that can then be accessed (read or updated) locally, thus improving response times. The goal here is not to deal with concurrent transactions and update conflicts but to avoid the large penalty of remote data access. When decrementing the value, if the site does not locally have the necessary amount, it will have to either abort the transaction or reconfigure, updating reserves from others. This technique has been generalized to enforce abstract predicates instead of simple lower bounds [29, 33, 66, 67]. There are two main fundamental differences in our proposal: First, distributed reservation systems are not concerned with routing clients to splits, as there is a trivial choice of using the local one. In contrast, the randomized method used in MRVs is a key part of our contribution. Second, in the distributed systems that we target, nodes do not own rows exclusively. They are either fully replicated (e.g., MySQL GR, MongoDB Replica Set) or hosted in a separate storage layer (e.g., Spanner, Aurora), hence, all splits are accessed equally efficiently by all clients and always lead to remote communication for commit, even if stored in private rows or tables. This means that, unlike MRVs, these techniques cannot be easily layered by application developers on top of existing systems as they would need changes to server source code or be infeasible in the case of cloud services. Simply using multiple server instances in each node to deal with concurrent updates is also not an option, as this would not be transparent to clients and would lead to a large space overhead as the number of splits could not be adjusted separately for each row.

Finally, there are numerous techniques based on replication and commutativity targeted at distributed systems, mainly for availability in a partitionable system [9]. Like MRVs and *phase reconciliation*, Cassandra's counters [65] are physically split into multiple structures (shards). Each site applies updates to one of them and the total is computed with a sum. *Delta transactions* [57] aims to convert two transactions that would otherwise conflict into multiple that commute, for example, by updating not *to* some value but *by* some value. Another approach to avoid conflicts of concurrent operations is to transform the operations themselves to achieve the same final state

across replicas [16]. *Conflict-free counting set objects* [56] generalize these techniques to sets and *Conflict Free Replicated Data Types* (CRDTs) [27, 51, 52] to a variety of data structures. These have been widely used to improve the availability of distributed databases [1, 32, 47, 50, 55]. Since some operations cannot commute (e.g., update with delete), some propose discarding one update in favor of the other when merging data replicas. A custom merge procedure can be used [59], but the most common conflict-solving rule is the *last writer wins* [60], used in various systems [2, 13, 30–32, 34, 49]. However, none of these solutions provides strong consistency: They are unable to enforce lower a bound invariant and in some cases arbitrarily discard data and operations, which we assume that it is not acceptable. The exception is the Bounded Counter CRDT [4], which employs *escrow reservations* [42, 43].

In short, MRVs are the only solution that enforces limit invariants – unlike most solutions based on commutativity – are viable in distributed systems – unlike *escrow locking* – and maximize parallelism - unlike RedBlue/$\mathcal{I}$-confluence, where *sub* operations with lower limit invariants are not commutative.

7 CONCLUSIONS AND FUTURE WORK

We address a classical problem in transactional data processing that has resurfaced in modern distributed database systems: Mitigating the effects of update conflicts on usable throughput while at the same time preserving strong isolation criteria. The novelty of our proposal, Multi-Record Values (MRVs), hinges on: The use of randomness to coordinate access to a value partitioned across multiple records, without expensive synchronization; and an implementation strategy that leverages existing transactional and query processing mechanisms of existing database management systems.

We then demonstrate the usefulness of the technique in a variety of database management systems (PostgreSQL, DBx1000, MongoDB, MySQL GR, and a cloud service) and standard benchmark workloads (TPC-C and STAMP Vacation), in all cases achieving at least 3× higher throughput in high-contention scenarios, and up to 100× higher in microbenchmarks. This is competitive with existing state-of-the-art in centralized systems and unprecedented in distributed systems.

By being easily implemented on client APIs and without changes to the database engine, MRVs can be readily deployed on existing systems, even on closed-source cloud ones. This also makes it easy to combine with state-of-the-art high-performance concurrency control to better cope with extreme throughput environments.

The question that now remains is whether the MRVs technique can be applied to other data structures. Since splitting-based techniques have previously been applied to non-numerical structures, such as top-k sets [36], future work can study the possibility of using MRVs in different contexts.

ACKNOWLEDGMENTS

Special thanks to Nuno Preguiça and the anonymous reviewers for their feedback. Partially funded by project AIDA – Adaptive, Intelligent and Distributed Assurance Platform (POCI-01-0247-FEDER-045907) co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal.

REFERENCES

[1] D. Akkoorath, A. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. 2016. Cure: Strong semantics meets high availability and low latency. In *IEEE 36th Intl. Conf. on Distributed Computing Systems (ICDCS)*. IEEE, 405–414.

[2] J.C. Anderson, J. Lehnardt, and N. Slater. 2010. *CouchDB: The Definitive Guide: Time to Relax*. O'Reilly Media. https://books.google.pt/books?id=G4N-DPk9R5sC

[3] P. Bailis, A. Fekete, M. Franklin, A. Ghodsi, J. Hellerstein, and I. Stoica. 2014. Coordination avoidance in database systems. In *Proc. VLDB Endowment*, Vol. 8. 185–196.

[4] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *IEEE 34th Intl. Symp. on Reliable Distributed Systems (SRDS)*. 31–36. https://doi.org/10.1109/SRDS.2015.32

[5] D. Barbará-Millá and H. Garcia-Molina. 1994. The Demarcation Protocol: A Technique for Maintaining Constraints in Distributed Database Systems. *The VLDB Journal* 3, 3 (1994). https://doi.org/10.1007/BF01232643

[6] R. Bayer and E. Mccreight. 1972. Organization and Maintenance of Large Ordered Indexes. *Acta Inf.* 1, 3 (Sept. 1972), 173–189. https://doi.org/10.1007/BF00288683

[7] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proc. 1995 ACM SIGMOD Intl. Conf. on Management of Data*. Association for Computing Machinery. https://doi.org/10.1145/223784.223785

[8] P. Bernstein, V. Hadzilacos, and N. Goodman. 1987. *Concurrency control and recovery in database systems*. Vol. 370. Addison-Wesley Reading.

[9] Eric A Brewer. 2000. Towards robust distributed systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, Vol. 7. 343477–343502.

[10] Michael J Cahill, Uwe Röhm, and Alan D Fekete. 2008. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*.

[11] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (Aug. 2013), 1–22. https://doi.org/10.1145/2491245

[12] Oracle Corporation. 2022. MySQL 8.0 Documentation - 18.1.1.2 Group Replication. https://dev.mysql.com/doc/refman/8.0/en/group-replication-summary.html.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Operating Systems Review - SIGOPS*, Vol. 41. 205–220. https://doi.org/10.1145/1294261.1294281

[14] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *Proc. 2013 ACM SIGMOD Intl. Conf. on Management of Data*. 1243–1254.

[15] Nick Dimiduk. 2022. Orderly library. https://github.com/ndimiduk/orderly.

[16] C. Ellis and S. Gibbs. 1989. Concurrency control in groupware systems. In *Proc. 1989 ACM SIGMOD Intl. Conf. on Management of data*. 399–407.

[17] J. Faleiro, D. Abadi, and J. Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[18] J. Gray and A. Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[19] The PostgreSQL Global Development Group. 2021. PostgreSQL 14 Documentation - 13.2. Transaction Isolation. https://www.postgresql.org/docs/14/transaction-iso.html.

[20] The PostgreSQL Global Development Group. 2021. PostgreSQL 14 Documentation - 41.4. Rules on INSERT, UPDATE, and DELETE. https://www.postgresql.org/docs/14/rules-update.html.

[21] Z. Guo, K. Wu, C. Yan, and X. Yu. 2021. Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking. In *Proceedings of the 2021 International Conference on Management of Data*. Association for Computing Machinery, New York, NY, USA, 658–670.

[22] Y. Huang, W. Qian, E. Kohler, B. Liskov, and L. Shrira. 2020. Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endowment* 13, 5 (2020), 629–642.

[23] MongoDB Inc. 2021. MongoDB 5.0 Manual - Read Concern. https://www.mongodb.com/docs/v5.0/reference/read-concern/.

[24] MongoDB Inc. 2021. MongoDB 5.0 Manual - Read Preference. https://www.mongodb.com/docs/v5.0/core/read-preference/.

[25] MongoDB Inc. 2021. MongoDB 5.0 Manual - Transactions. https://www.mongodb.com/docs/v5.0/core/transactions/.

[26] MongoDB Inc. 2021. MongoDB 5.0 Manual - Write Concern. https://www.mongodb.com/docs/v5.0/reference/write-concern/.

[27] M. Letia, N. Preguiça, and M. Shapiro. 2010. Consistency without concurrency control in large, dynamic systems. *Operating Systems Review* 44 (04 2010), 29–34. https://doi.org/10.1145/1773912.1773921

[28] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. 2012. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI 12)*. 265–278.

[29] J. Liu, T. Magrino, O. Arden, M. George, and A. Myers. 2014. Warranties for faster strong consistency. In *11th USENIX Symp. on Networked Systems Design and Implementation (NSDI 14)*. 503–517.

[30] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proc. 23rd ACM Symp. on Operating Systems Principles*. 401–416.

[31] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen. 2013. Stronger semantics for low-latency geo-replicated storage. In *Proc. 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI 13)*. 313–328.

[32] P. Lopes, J. Sousa, V. Balegas, C. Ferreira, S. Duarte, A. Bieniusa, R. Rodrigues, and N. Preguiça. 2019. Antidote SQL: Relaxed When Possible, Strict When Necessary. *CoRR* abs/1902.03576 (2019). arXiv:1902.03576 http://arxiv.org/abs/1902.03576

[33] T. Magrino, J. Liu, N. Foster, J. Gehrke, and A. Myers. 2019. Efficient, consistent distributed computation with predictive treaties. In *Proc. 14th EuroSys Conf. 2019*. 1–16.

[34] Microsoft. 2021. Microsoft Azure CosmosDB Documentation - Conflict types and resolution policies when using multiple write regions. https://docs.microsoft.com/en-us/azure/cosmos-db/conflict-resolution-policies.

[35] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization*. IEEE, 35–46.

[36] N. Narula, C. Cutler, E. Kohler, and R. Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 511–524.

[37] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (jun 1996), 351–385. https://doi.org/10.1007/s002360050048

[38] Patrick E O'Neil. 1986. The escrow transactional method. *ACM Trans. on Database Systems (TODS)* 11, 4 (1986), 405–430.

[39] P. Peinl, A. Reuter, and H. Sammer. 1988. High contention in a stock trading database: a case study. In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data* (Chicago, Illinois, USA) *(SIGMOD '88)*. Association for Computing Machinery, New York, NY, USA, 260–268.

[40] Percona-Lab. 2022. Sysbench-tpcc source code. https://github.com/Percona-Lab/sysbench-tpcc/.

[41] D. Ports and K. Grittner. 2012. Serializable Snapshot Isolation in PostgreSQL. (Aug. 2012). arXiv:1208.4179 [cs.DB]

[42] N. Preguiça, J. Martins, M. Cunha, and H. Domingos. 2003. Reservations for conflict avoidance in a mobile database system. In *Proc. 1st Intl. Conf. on Mobile systems, applications and services*. 43–56.

[43] N. Preguiça, C. Baquero, F. Moura, J. Martins, R. Oliveira, H. Domingos, J. Pereira, and S. Duarte. 2000. Mobile Transaction Management in Mobisnap. In *Proc. East-European Conf. on Advances in Databases and Information Systems (with Intl. Conf. on Database Systems for Advanced Applications: Current Issues in Databases and Information Systems) (ADBIS-DASFAA '00)*. Springer-Verlag, Berlin, Heidelberg, 379–386.

[44] A. Prout, S. Wang, J. Victor, Z. Sun, Y. Li, J. Chen, E. Bergeron, E. Hanson, R. Walzer, R. Gomes, and N. Shamgunov. 2022. Cloud-Native Transactions and Analytics in SingleStore. In *Proceedings of the 2022 International Conference on Management of Data*. 2340–2352.

[45] L. Qu, Q. Wang, T. Chen, K. Li, R. Zhang, X. Zhou, Q. Xu, Z. Yang, C. Yang, W. Qian, and A. Zhou. 2022. Are current benchmarks adequate to evaluate distributed transactional databases? *BenchCouncil Transactions on Benchmarks, Standards and Evaluations* 2, 1 (March 2022), 100031.

[46] Francois Raab. 1993. TPC-C – The Standard Benchmark for Online transaction Processing (OLTP). *The Benchmark Handbook* (1993).

[47] Redis. 2020. Redis Blog - Diving into CRDTs. https://redis.com/blog/diving-into-crdts/.

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

[49] Riak. 2017. Riak KV 2.2.3 Documentation - Conflict Resolution. https://docs.riak.com/riak/kv/2.2.3/developing/usage/conflict-resolution/index.html.

[50] Riak. 2017. Riak KV 2.2.3 Documentation - Data Types. https://docs.riak.com/riak/kv/2.2.3/learn/concepts/crdts.

[51] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. Conflict-free replicated data types. In *Symp. on Self-Stabilizing Systems*. Springer, 386–400.

[52] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[53] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. 1995. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)* 20, 3 (1995), 325–363.

[54] L. Shrira, H. Tian, and D. Terry. 2008. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In *ACM/IFIP/USENIX Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*. Springer, 42–61.

[55] Dharma Shukla. 2018. Azure CosmosDB: Pushing the frontier of globally distributed databases. https://azure.microsoft.com/en-us/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/.
[56] Y. Sovran, R. Power, M. Aguilera, and J. Li. 2011. Transactional storage for geo-replicated systems. In *Proc. 23rd ACM Symp. on Operating Systems Principles (SOSP)*. 385–400. https://doi.org/10.1145/2043556.2043592
[57] Dan Stocker. 2010. Delta Transactions. https://collectiveweb.wordpress.com/2010/03/01/delta-transactions/.
[58] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, L. Zhang J. Jaffray, and P. Mattis. 2020. CockroachDB: The resilient geo-distributed SQL database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1493–1509.
[59] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proc. 15th ACM Symp. on Operating Systems Principles*. https://doi.org/10.1145/224056.224070
[60] Robert H. Thomas. 1979. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. on Database Systems (TODS)* 4, 2 (1979), 180–209.
[61] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. 2013. Speedy transactions in multicore in-memory databases. In *Proc. 24th ACM Symp. on Operating Systems Principles*. 18–32.
[62] A. Verbitski et al. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proc. 2017 ACM Intl. Conf. on Management of Data*. 1041–1052. https://doi.org/10.1145/3035918.3056101
[63] W. Wang, M. Hsu, and E. Pinsky. 1991. Modeling hot spots in database systems. In *Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems - PODS '91* (Denver, Colorado, United States). ACM Press, New York, New York, USA.
[64] C. Yao, D. Agrawal, G. Chen, Q. Lin, B. Ooi, W. Wong, and M. Zhang. 2016. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Transactions on Knowledge and Data Engineering* 28, 10 (2016), 2635–2650.
[65] Aleksey Yeschenko. 2014. The DataStax Blog - What's New in Cassandra 2.1: Better Implementation of Counters. https://www.datastax.com/blog/whats-new-cassandra-21-better-implementation-counters.
[66] H. Yu and A. Vahdat. 2000. Design and evaluation of a continuous consistency model for replicated services. In *Proc. 4th Symp. on Operating System Design & Implementation-Volume 4*.
[67] Ha. Yu and A. Vahdat. 2000. Efficient Numerical Error Bounding for Replicated Network Services. In *Proc. 26th Intl. Conf. on Very Large Databases (VLDB)*.
[68] Xiangyao Yu et al. 2022. DBx1000 source code. https://github.com/yxymit/DBx1000/.
[69] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endowment* 8, 3 (nov 2014), 209–220. https://doi.org/10.14778/2735508.2735511
[70] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. 2016. Tictoc: Time traveling optimistic concurrency control. In *Proc. 2016 Intl. Conf. on Management of Data*. 1629–1642.

Received April 2022; revised July 2022; accepted August 2022