Diverse Database Replication Based on Snapshot Isolation – Performance Implications of Improved Dependability

Peter Popov and Vladimir Stankovic
Centre for Software Reliability, Department of Computer Science, City St George's, University of London
Email: P.T.Popov@citystgeorges.ac.uk, Vladimir.Stankovic.1@citystgeorges.ac.uk

***Abstract*—Numerous database replication schemes are built on the crash failure assumption where majority of failures are self-evident as defined in [1]. The study in [1] convincingly refuted this common assumption showing that many of the faults in relational Database Management Systems (DBMSs) cause systematic non-crash failures. Similar results were obtained in the subsequent study [2]. Consequently, the existing database replication solutions, which typically use the same DBMS, are ineffective fault-tolerant mechanisms. Conversely, using diverse DBMSs is a suitable way of protecting against non-crash failures.**

We have built a middleware-based database replication protocol, *DivRep*, and deployed it with diverse database servers (*DivSQL*), for improved fault tolerance. DivSQL provides *strict* Snapshot Isolation (SI) guarantees, and assumes "incorrect results" failure model (IRFM) – the most realistic one based on the extensive experimental analyses of DBMS faults ([1, 2]). The dependability gain comes with the inherent performance overhead. We provide a comprehensive performance evaluation of DivSQL using 3 diverse DBMSs (two are leaders in the field).

Keywords—database replication, dependability, design diversity, performance evaluation.

I. Introduction

Database replication has proved as a practical way of improving both dependability and performance. Traditional replication schemes based on Read-Once-Write-All-Available (ROWAA) approach [3] provide increased availability and guard against crash failures by deploying non-diverse solutions, i.e., using relational Database Management Systems (DBMSs) from a same vendor. The scalability is typically improved by balancing the load among the deployed replicas.

However, two extensive studies with diverse DBMSs [1, 2] showed that a significant proportion of software faults ("bugs") cause non-crash failures. In [1] a majority of the reported faults for all 4 DBMSs studied led to "incorrect result" failures rather than crashes (about 65% vs. 18%). Similar results were obtained in [2]: over 50% of the reported bugs caused non-crash failures in the 3 DBMSs tested – mostly incorrect answers or database corruptions. The fact that bugs leading to non-crash failures are difficult to detect, and thus probably underreported, is likely to have contributed to the view that they are not a major problem. It also implies that the reported figures probably underestimated the true fraction of faults that cause non-crash failures.

The findings in [1] demonstrated that very few cases that triggered a bug in one DBMS caused failures in another. They showed that a pair of diverse servers achieves failure detection rate of *no less* than 95%. Moreover, there were no coincident failures in more than two of the servers. The results warrant the choice of *design diversity*, i.e., *diverse redundancy* (when tolerating design faults via deploying diverse (DBMS) products in parallel), as a suitable way for improving dependability.

We have built a middleware-based database replication scheme, DivSQL, which ensures *strict* consistency levels: both *strict 1-copy snapshot* isolation (the correctness criterion based on *1-copy snapshot isolation*, 1-copy SI, [4]) and *Conventional Snapshot Isolation (CSI)* [5] as seen in centralised DBMSs. The latter is referred to as just Snapshot Isolation (SI) [6]. DivSQL offers dependability assurance against software faults, i.e., high failure detection rate, achieved by deploying two diverse DBMSs. We use a leading commercial DBMS – *Comm* for short, and open-source products: Firebird[1] (*FB*) and PostgreSQL (*PG*). The replication scheme is configurable to different *regimes of operation* depending on the specific client requirements, optimized for either dependability assurance (*pessimistic* regime) or performance improvement (*optimistic* regime). To improve fault tolerance, the pessimistic (*Pess*) regime features a *Comparator* component that compares, within the transaction boundaries, the results of SQL operations coming from different DBMSs. The optimistic (*Opt*) regime implements a *skip* feature, which improves performance by executing read operations only on one of the DBMSs. The two regimes are complementary and can be combined into a configurable quality of service, i.e., a *hybrid* regime of operation[2] (Sect. VI.B).

Intuitively, the application of design diversity for improving dependability, e.g., through fault tolerance, is likely to exhibit performance cost, which is precisely the focus of this paper. We compared the performance of *Pess* and *Opt* regimes of operation against single DBMSs, and non-diverse pairs. The results are encouraging: when the individual performances of diverse DBMSs are close, *Pess* regime is no more than 6% worse than the faster single server, and *Opt* regime exhibits higher throughput than the faster single server for some loads. Also, the performance results of the two regimes estimate the bounds of the performance attainable with the *hybrid* regime of operation.

[1] Firebird is the open-source descendent of InterBase, one of the first DBMSs to implement Multi Version Concurrency Control (MVCC) – see the seminal work on Snapshot Isolation [6] and here.

[2] The *hybrid* regime of operation has been conceived, but not yet implemented.

DivSQL is deployed with a pair of diverse replicas (e.g., 1FB and 1PG DBMS) in a single fault–tolerant node (DivSQL), which is sufficient for achieving improved *detection* of non-crash failures. Should a higher level of replication be required, e.g., for better scalability, then DivSQL can be combined with another database replication scheme, which is considered adequate for a particular set of requirements. These can be schemes optimised for scalability; be it eager database replication, e.g., based on group communication primitives, or lazy replication. In either case, DivSQL would replace a non-diverse replica used by the chosen database replication scheme. Such a "fusion" replication scheme will allow one to have both high dependability assurance (via DivSQLs) and improved performance (via the chosen scalable replication).

The main *contributions* of the paper are as follows: a) we provide a novel middleware-based database replication protocol for SI-based DBMSs, and proofs of its strict consistency; b) this is the first (diverse) database replication approach that assumes a *realistic*, *experimentally-confirmed* failure model ("incorrect results" failure model – IRFM) which lies between the two most studied ones: crash, and Byzantine, failure model, and c) this is the first paper to provide comprehensive experimental evaluation of an SI-based database replication with *diverse* relational DBMSs (two of the chosen products are the leaders in the field).

The rest of the paper is organized as follows. We review related work about database replication and design diversity in Sect. II. Sect. III explains the replication model and provides necessary definitions. Sect. IV describes our replication protocol. In Sect. V we show the extensive experimental evaluation of DivSQL performance against the single DBMSs and non-diverse pairs. In Sect. VI we discuss the findings, and finally in Sect. VII we present conclusions and state provisions for future work.

II. RELATED WORK

A. Database Replication

There exist numerous database replication solutions, both commercial and academic. These solutions have been categorized in different ways, e.g. *middleware-based* (black-box) vs *kernel-based* (white-box) approaches (grey-box solutions, whereby some internals of DBMSs are made use of, exist too); based on *where* the transactions are executed (primary-backup vs. multi-master approaches), or *when* the transactions are executed on replicas (*eager* vs *lazy*) [7] [8]; either a *full* (every node stores a copy of all data items) or a *partial* replication (each node has a subset of data items), etc.

Most database replication solutions, in academia and industry, are based on crash failure model. The one from [4] is a middleware-based solution, and it defines the correctness criterion based on SI – *1-copy snapshot isolation*, a strict version of which DivSQL ensures.

There are, however, few research works on database replication that assume Byzantine failure model, [2], [9], [10], [11], [12]. In [2], HRDB, unlike our solution, provides 1-copy serializable guarantees. HRDB is a primary copy replication scheme and thus the performance of the replicated system is dictated by the processing of the primary. The main premise is Commit Barrier Scheduling (CBS), which allows the Shepherd to execute transactions on the primary and ensure that all non-faulty secondaries execute transactions in an equivalent serial order. HRDB deals with the non-determinism of the locking mechanisms in diverse replicas by requiring the primary is *sufficiently blocking* – if the primary executes a pair of SQL operations in parallel, it is ensured all non-faulty secondaries can do so, too. The authors place trust into the Shepherd component, which is also a single point of failure. The authors report HRDB overhead of 17% compared to single server using TPC-C workload, but use non-diverse servers (MySQL). Analogous to CBS, Snapshot Epoch Scheduling (SES) protocol for SI DBMSs is given in [9]. The evaluation is based on their own benchmark – the results for loads up to 20 Clients are good, after which performance degrades. The evaluation against TPC-C is not given due to implementation issues (Sect. 6.4 in [9]).

The work in [10] is similar to ours in that DBMSs offering SI are used, but Byzantine failure model is assumed. Notably, in their performance analysis of Byzantine behavior (Sect. 7.4) they used an "incorrect result" failure (DivSQL guards against those) and justify the choice by the failure being a more realistic one than a more demanding Byzantine failure. The authors do not deploy diverse DBMSs for their performance evaluation; they use a single DBMS product – PostgreSQL. The reported overhead (20% to 35%) for the write-intensive workload (TPC-C), as well as the improved performance for the read-only workload (using only read-only transaction types from TPC-C), when compared to the single DBMS is not sufficiently realistic – the inevitable differences in the individual performances of diverse DBMSs would have likely impacted these results, and thus the conclusions made likely do not apply more generally.

In [11], the authors propose a Byzantine fault-tolerant deferred update replication protocol, and consider a set of Byzantine client attacks (on the certification step of the protocol) for which they provide countermeasures and a simulation model to evaluate them. They do not provide (experimental) evaluation of the protocol, however.

In the early solution of Byzantine fault-tolerant databases [12] all operations are serialised: this provides high consistency level at the major expense of precluding concurrent transactions.

B. Design Diversity

Design diversity was conceived more than 40 years ago, and its use to improve fault tolerance has been researched extensively. A survey of effectiveness of design diversity can be found in [13], and its design aspects can be found in [14]. Due to its high cost the industry has been reluctant to use design diversity for guaranteeing sufficient levels of dependability except for safety-critical applications e.g., protection systems in nuclear industry, avionics, etc. Proliferation of off-the-shelf software gave rise to the use of diversity as a realistic possibility for dependability improvement. However, the results of assessing the potential dependability gains, and more so the consequent performance overhead, are still scarce, and especially regarding the software we are interested in – DBMSs.

Gashi et al. [1] discuss architectural issues and difficulties that arise when using design diversity in the context of DBMSs. Use of rephrasing rules, as a "data diversity" mechanism, for failure diagnosis and state recovery was studied in [15].

III. Replication and System Model

Each DivSQL contains two DBMS replicas (we use terms 'DBMS' and 'replica' synonymously in the rest of the paper). DivSQL uses *full, middleware-based* replication and guarantees data consistency by enforcing *eager, multi-master* approach [7]. It assumes IRFM failure model. The replication middleware supports *interactive transactions* and replication is done at the level of SQL operations. Hence, runtime dependence between SQL operations within a transaction – a realistic, and favorable trait, since in general SQL operations are not known at the transaction start – is allowed. Clients wait for response to an operation before sending the next one. DivSQL implements *active* replication [16] – all SQL operations are executed on both replicas in the *Pess* regime, which enables a straightforward comparison of diverse DBMS results. DivSQL employs *DivRep* replica control protocol. Fig. 1 shows DivSQL architecture.

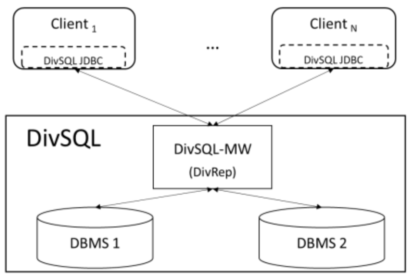

Fig. 1 DivSQL system architecture.

Database transactions are sequences of *read* (r) and *write* (w) operations and each transaction starts with a *begin* (b) operation and ends successfully with a *commit* (c) or must *abort* (a) and roll back to the state before it started. Begins, commits and aborts are all *transaction boundary* operations.

We assume that each DBMS offers *snapshot isolation* (SI) [6]. A transaction executing in SI operates on a snapshot of committed data, taken upon the transaction's begin. Hence, SI guarantees that all reads of a transaction see a consistent snapshot of the database. The changes performed during a transaction are seen by the subsequent reads within that same transaction. SI does not exhibit any of the anomalies from the SQL standard (*Dirty Read, Non-repeatable Read*, or *Phantom Read*). The concurrency control mechanism of a DBMS implementing SI raises an exception only due to *write-write* conflicts. These conflicts occur when concurrent transactions attempt to write to the same data item (e.g., a table row). Two transactions, T_i and T_k, are concurrent if the following holds: T_i*Begin* < T_k*Commit* < T_i*Commit*. The absence of conflicts between readers and writers in SI improves performance and makes it more appealing than the traditional *serializable* isolation level. SI is offered in leading commercial and open-source relational DBMSs (MSSQL, Oracle, PostgreSQL etc.), as well as in some NoSQL ones, e.g., MongoDB. There exist solutions to change database application executing under SI in such a way that serializability is guaranteed [17]. Also, executing many typical workloads, including several TPC benchmarks, under SI provides the same results as if they were executed under serializable isolation level.

Real-world DBMSs implement SI using write locks together with multi-versioning, instead of, for example, an optimistic concurrency control mechanism that checks for *write-write* conflicts in the end of transactions (thus, use First-Updater-Wins [18] rule rather than First-Committer-Wins). Our approach assumes the former.

We distinguish two types of transactions: read-only, which have only read operations, and write transactions that have at least one write and zero or more read operations.

Some academic proposals for database replication (e.g. [2], [9], [10], [11]) considered Byzantine failure model for DBMSs – the most general model in which components of a system fail in arbitrary ways (in addition to crashing they can process requests incorrectly, corrupt their state, produce inconsistent outputs due to malicious intent, etc.). Non-crash failure models must be addressed, and we have contributed to the development of novel protocols that consider those. However, the *IRFM failure model* assumed by DivSQL lies between crash and Byzantine failures models. In addition to crash failures, it guards against failures of SELECTs (e.g., erroneous omission of a row) – referred to by some as "logic bugs" [19], and DELETEs, INSERTs and UPDATEs wrongly changing the database state. It is likely the most suitable failure model for majority of applications based on the compelling evidence that non-crash failures are dominant in DBMSs [1, 2]. Our approach is thus based on a failure model with practical relevance. Further research is needed to establish how likely DBMSs and the clients exhibit more challenging Byzantine behaviour.

Client applications access our system using JDBC interface. Our JDBC driver implements the client side of DivRep protocol. We use different SI-enabled DBMSs, and their JDBC drivers, in DivSQL. We configured our prototype implementation to run one of the 3 pairs: 1Comm1FB, 1Comm1PG or 1FB1PG.

IV. Diverse Replication Protocol (DivRep)

A. *DivRep Architecture*

A (diverse) database replication protocol – *DivRep* – ensures data consistency in DivSQL. DivRep uses mechanisms directly supported by off-the-shelf DBMSs. Thus, our solution can be an add-on, does not require changes of off-the-shelf DBMSs or to provide extensions that duplicate a core functionality of the existing products as is the case with some replication protocols.

For every client connected to a DivSQL a Transaction Manager (*TraManager*) is provided. In turn, every TraManager communicates to the database replicas via a replication manager (*RepManager*), one for each DBMS. Since we envisage DivSQL to consist of a pair of DBMSs, there are 2 RepManagers per every TraManager. Fig. 2 presents a pseudocode of DivRep executing in the *Pess* regime in a TraManager, while Fig. 3 does so for a RepManager. The correctness of DivRep is ensured when an arbitrary number of replicas are deployed, not only two. Thus, we use the pronouns "all" and "both" interchangeably.

A TraManager accepts operations submitted by a particular client. It deals in a specific way with every operation depending on its type, e.g., transaction boundaries are treated differently than the reads and writes. Transaction execution occurs in parallel on the two DBMSs.

The execution of a *begin* operation is as follows: first, the variable indicating that a particular transaction should abort is

```
I)  Upon SQL operation OP from Ti
II) switch(OP)
    A)  case: OP is a begin operation
        1)  reset transaction abort /* set transaction abort to false */
        2)  obtain the mutex /* among all TraManagers */
        3)  send begin to all DBMSs /* create snapshots on the DBMSs */
        4)  wait until all DBMSs begin the transaction
        5)  release the mutex
        6)  return control to the client

    B)  case: OP is a read or a write operation
        1)  put OP into the queues of RepManagers /* each DBMS is served by a RepManager */
        2)  receive the faster response for the OP
        3)  if the faster response is not an exception
            a)  return the response to the client
        4)  else
            a)  set transaction abort /* further execution of the transaction is prevented */
            b)  notify the client of the exception
        5)  wait for all responses to be received
            a)  if exception raised && transaction not set to abort
                i)      set transaction abort /* further execution of the transaction is prevented */
                ii)     notify the client of the exception
            b)  else
                i)      do compare responses /* start Comparator function */

    C)  case: OP is an abort operation
        1)  set transaction abort /* further execution of the transaction is prevented */

    D)  case: OP is a commit operation
        1)  if transaction not set to abort
            a)  wait until all available replicas "vote" (complete all operations) and the Comparator
                "votes" (compares all operations' responses)
                i)      obtain the mutex
                ii)     send commit to all DBMSs /* directly access their SQL API */
                iii)    wait until all DBMSs commit the transaction
                iv)     release the mutex
                v)      clear RepManagers queues
                vi)     return control to the client

   Abort Function /* executed by a TraManager */
I)  if transaction set to abort
    A)  abort transaction on all DBMSs /* directly access their SQL API */
    B)  wait until all DBMSs abort the transaction

   Comparator Function /* executed by a TraManager asynchronously (e.g., in a separate thread) */
I)  compare the responses from all DBMSs
    A)  if a mismatch found
        1)  set transaction abort /* further execution of the transaction is prevented */
        2)  notify the client of an exception ("data inconsistency occurred")
```

Fig. 2 The pseudo code of a TraManager execution in DivRep in *Pess* regime

reset, i.e., its value is set to false; then the `global mutex`, contended for by all TraManagers, is acquired; the begin is sent to all DBMSs for transaction snapshots to be created – this is done directly through each replica's SQL API, without sending it first to RepManagers. *No commit or begin* operation can execute unless the TraManager holds the `mutex`, and thus consistent snapshots (unchanged by any other transaction commit) are taken on both replicas; the `mutex` is released; finally, the control is returned to the client. If an exception is raised during the processing of a begin, the transaction abort is flagged, which subsequently triggers the *Abort function* (Fig. 2). The `mutex` could be replaced with an atomic broadcast primitive, or Paxos commit [20], but such a choice would introduce an unnecessary complexity for the two-channel replication proposed by DivSQL.

The execution of *reads* and *writes* is treated in the same way in TraManager. First, the operation is placed in the queues of both RepManagers. Once the faster response is received (i.e., a DBMS has successfully executed the operation, passed to it by the respective RepManager (Fig. 3)), it is returned to the client – this reduces SQL operation latency observed by the client. The client then sends the following operation, possibly using the results of the previous one. Thus, DivSQL allows dependence among SQL operations – interactive transactions.

If an exception is received, however, the transaction is set to abort, triggering the Abort function. Without occurrence of an exception, the processing continues by TraManager waiting for the responses from both DBMSs. Once both are collected, the *Comparator function* is initiated (Fig. 2) in the *Pess* regime. Like the Abort function, the Comparator executes asynchronously with TraManager and the respective RepManagers. It first does

```
I)  while (non-empty queue && transaction not set to abort)
    A)  fetch an operation (OP) from the RepManager queue
    B)  switch(OP)
        1)  case: OP is a read operation
            a)  send OP to the respective DBMS
            b)  wait for a response /* either fetched data or an exception */
            c)  return response to the TraManager
        2)  case: OP is a write operation
            a)  send OP to the respective DBMS
            b)  wait for a response /* data or an exception */
            c)  if no exception raised
                i)  obtain the writeset - send respective control read /* this operation may result in an
                    exception, which would be stored in the writeset itself */
                ii) return the writeset to the TraManager
            d)  else
                i)  return exception to TraManager /* write-write conflict */
        3)  default: /* unsupported operation */
```

Fig. 3. The pseudo code of a RepManager execution in DivRep in *Pess* regime

metadata checks (e.g., number of rows), and then pair-wise comparison, value by value, of the corresponding DBMS responses. If it finds a mismatch the transaction abort is set, indicating to the client that "data inconsistency" exception occurred. To compare the effects of write operations the middleware generates *control read (control SELECT)* operations. A control read is constructed by parsing the respective write operation and it is sent straight after the write to retrieve the modified tuples, and no others. Since the underlying DBMSs offer SI and execution order of boundary operations is the same on both servers, the algorithm provides necessary replica determinism. If no failure occurs the replicas produce the same results. The result comparison of *all* operations (reads, control reads, and writes for which the number of modified rows is usually returned to the client) is completed before the commit.

The *control reads* impose more roundtrips between the middleware and the replicas. This could be alleviated using e.g., SQL extensions for data-change operations (DELETE, INSERT, UPDATE) as in DB2 [21], which return a result set with modified rows. Alternatively, already fully-implemented approaches *might* be used: writeset extraction via triggers or transaction log, or the RETURNING clause in PG or the OUPUT clause in MSSQL. But these will, at best, need special treatment due to the use of diverse DBMSs.

The transaction duration is likely to be prolonged due to the use of the Comparator function. The actual delay is, however, likely to be negligible because the results' comparison in DivRep is performed in parallel with the operations' executions on the DBMSs. Thus, in many cases, all except the result of the last operation will have been compared before the slower server completes all operations. The assumption of the minimal delay depends on the performance characteristics of the Comparator and the sizes of the results. Hashing the results of (control) reads could minimize processing time needed for comparison.

Once the *commit* operation is submitted, TraManager checks if a transaction abort has been already set. If it has not, once the "votes" from all replicas (confirming that all reads and writes have finished) and the Comparator "vote" (confirming no result inconsistency was found) are collected, the `mutex` is acquired. Like with a begin operation execution, no commit or begin from other transactions can execute while the TraManager holds the `mutex`. This guarantees the order of the commits and begins is the *same* on all replicas. After all replicas have acknowledged that the commit has been executed, the `mutex` is released. The queues of the RepManagers are then cleared, preparing them for the execution of the next transaction, and the control is returned to the client. Thus DivRep executes a variant of atomic commit ([22]) protocol (AC-DR): once both replicas and the Comparator have "voted", DivRep ends the transaction on both replicas. A replica either successfully completes all operations in a transaction or it raises an exception. The middleware regards the former as a "vote" [22] for *commit* and the latter as an *abort* "vote". Likewise, only if Comparator reports no inconsistencies between the respective results, a *commit* "vote" is recorded.

If an exception occurs during the processing of a transaction, TraManager notifies the client that the transaction needs to abort. As a result, the Abort function is triggered by the client (in typical use of transactions this is indeed client's responsibility) – the function submits aborts to all DBMSs through the respective SQL API without sending them via the RepManagers (abort is executed asynchronously for each DBMS). If the client submits an abort operation as part of its workflow, the TraManager triggers the Abort function by setting transaction abort, too.

To minimise the performance overhead DivRep aborts the transaction on both replicas as soon as a write-write conflict or a result inconsistency is reported, instead of performing agreement phase once both replicas have finished all SQL operations and the comparison of the results has been completed. It is possible that the replica on which the write-write conflict was not raised is still executing an operation, and the RepManager is therefore blocked. In this case, one could cancel the execution of the currently running operation on the non-aborting DBMS. The trade-off between successful cancelling of a running SQL operation and performance benefits it can bring is mute, since the action must be done asynchronously. Also, the cancel functionality must be supported by both the DBMS and the connectivity layer. We have not implemented the cancel feature.

The part of DivRep protocol executed on RepManagers (Fig. 3) is simpler than the TraManager execution – RepManagers execute only read and write operations. While there are unexecuted operations in a particular RepManager queue and the corresponding transaction is not set to abort, the processing proceeds as follows. If it is a read, the operation is sent to the DBMS, the response (either data or the exception due to unsuccessful execution) is fetched and returned to the TraManager. The processing of a write operation differs from the processing of a read in that RepManager must explicitly initiate the extraction of the writeset using control reads. The writeset, identically to the result of a read operation, is sent to the Comparator function of the TraManager for validation (the Comparator runs in a separate thread). If a bug manifests itself while a control read is executed (see I)B)2)c) in Fig. 3), DivRep treats it as any other failure, aborts the transaction to ensure consistency on the two DBMSs.

In practice SI is implemented using locks for write operations (Sect. III). Since we use multi-master approach, and transactions execute concurrently on the DBMSs in DivSQL, different orderings of (conflicting) SQL operations may ensue, and possibly lead to a distributed deadlock (the replicas must, in any case, agree on the transaction outcome via the atomic commitment). Thus, an integral part of DivRep is *NOWAIT* configuration parameter, set on a DBMS, and used for resolution of distributed deadlocks. NOWAIT exception is raised as soon as the DBMS detects that a transaction attempts to modify a data item for which an incompatible lock is held by a concurrent transaction. The feature is typically implemented as part of the first phase of a 2-Phase Locking protocol. The use of NOWAIT resolves consistently the consequences of non-determinism between the DBMSs. Many DBMSs (e.g., Firebird, MSSQL, Oracle, PostgreSQL, etc.) offer NOWAIT functionality, though the respective implementations and intended use differ (e.g., NOWAIT can be configured on the level of the DBMS, DB connection, or SQL operation).

NOWAIT is enabled on *only one* DBMS in DivRep. This asymmetric configuration of NOWAIT precludes inconsistent decision by NOWAIT on different DBMSs. DivRep uses a NOWAIT feature to immediately report the write-write conflicts raised by one of the DBMSs, not both. Thus, most write-write conflicts will typically be reported by NOWAIT-enabled DBMS while on the other replica transaction blocking might ensue. Also, potentially high abort rate is curtailed by setting NOWAIT only on one DBMS. Whether NOWAIT exception will be raised or not depends on the individual speed of, and transaction ordering on, the two DBMSs. It is possible that the DBMS with NOWAIT disabled raises a write-write exception – these exceptions will trigger transaction aborts by DivSQL too, and consistent state across the DBMSs will be preserved.

It is evident that DivSQL represents a single point of failure. Standard techniques, such as primary-backup replication [23] or a decentralised DivSQL could alleviate this problem. DivSQL is relatively simple and, thus, we have high confidence in its being implemented correctly, i.e., free of design faults. The effects of other faults, e.g., that can be attributed to hardware issues, are adequately modelled as crashes. Assuming crash behaviour for DivSQL, therefore, becomes plausible, thus making DivSQL suitable for integration into a highly scalable, and/or highly available, replication solution – the "fusion" approach (Sect. I).

B. DivRep Optimistic Regime

In *Opt* regime no adjudication of the responses from the diverse DBMSs occurs. Also, a *skip* feature is implemented in the middleware as follows. Before a replica (DBMS 1), executes a read operation, DivRep checks if a response to this operation has already been received from the other replica (DBMS 2). If so, then DBMS 1 does not execute the operation, i.e., skips it. The write operations are executed on all replicas, i.e., they cannot be skipped. The functionality of looking up the next operation and the *skip* feature is implemented in the RepManagers, which relays to the DBMSs the operations for execution. If a read operation is to be skipped, then the RepManager simply does not pass it to the respective DBMS for execution. Clearly, this regime does not offer the same dependability assurance as the *Pess* one. It may, however, be adequate in many cases.

There might exist *systematic* differences between the times it takes diverse DBMSs to execute the same operation, e.g., due to the respective execution plans being different, the concurrency control mechanisms being implemented differently, etc. In the *Pess* regime, the *skip* feature is not used and the best that DivSQL can do *during* transaction execution is to process SQL operations as fast as the faster of the two servers can. However, DivSQL waits for both servers to complete *all* operations and performs adjudication of the respective responses, and thus diversity cannot bring any performance gains. When the *Opt* regime is used, however, the systematic difference might lead to improved performance. If the mix of operations within a transaction is such that both servers 'skip' reads, then the transaction might take DivSQL less time than either of the DBMSs it consists of.

C. Correctness and Liveness of DivRep

DivRep ensures strict consistency – it produces transaction execution histories equivalent to a history of a centralized SI scheduler. DivRep guarantees *strict 1-copy SI*, a criterion based on 1-copy-SI [4], as well as *Conventional Snapshot Isolation* [5], the strictest SI level for (replicated or centralised) DBMSs. Respective proofs are given in the Appendix. Thus, DivRep provides (read-only) transactions with the most recent snapshot, a property commonly unavailable in other replication solutions, which permit stale data to be read. Informal reasoning about DivRep correctness and liveness follows.

First, the replicas start execution from the same state, i.e., the changes of all transactions that committed (the data item versions they installed) are visible upon a transaction begin by any client – this is ensured via using the `mutex` for all begins and commits.

Second, if two transactions (with conflicting writesets) overlap from the client's perspective they overlap in the global schedule produced by DivRep (all concurrent transactions on one DBMS are concurrent on the other, too), and if a transaction is aborted due to an exception raised (due to a write-write conflict, or else) the schedule observed on the client and the schedule produced by DivRep is the same, and all conflicting transactions ordered in the same way on both DBMSs. The execution from the client's perspective is the same as the execution in DivSQL. This resembles Commitment Ordering that like DivRep uses atomic commitment but ensures global serializability in replicated databases [24].

DivRep is correct under assumption that once a replica has voted for commit (II.D.1.a in Fig. 2), the commit is guaranteed to succeed on the given replica.

DivSQL ensures correctness in an asynchronous network model, and like in [10] we need a form of synchrony to guarantee liveness [25]. We assume *eventual* synchrony – DivSQL guarantees the clients make progress during periods when the delay to deliver a message is bounded.

V. Experimental Results

A. Test Harness and Implementation

We conducted extensive performance evaluation with DivSQL. We used 3 DBMSs (a commercial, and two open-source), 2 of which are leaders in the field: the commercial – *Comm*, Firebrid v5.0 – *FB* and PostgerSQL v17 – *PG*. Notably, our work has not aimed at performance ranking individual DBMSs, but instead focuses on evaluation of DivSQL: the intrinsic dependability assurance overhead of *Pess* regime, and the potential for performance improvement via *Opt* regime.

The DBMSs were deployed on ESXi Virtual Machines (VMs) with modest resources: 4 CPUs – 2.8GHz each, 16GB RAM, and 512GB SSD disks. The VMs ran Windows Server 2016 OS. The client application ran on one of the 4 VMs.

We implemeted DivSQL, and its JDBC driver, in Java (v23.0.1). In JDBC, SQL write operations (DELETEs, INSERTs, UPDATEs) return the number of modified rows. In *Pess* regime, comparison of the results from all SQL operation types is done: reads – result sets; writes – the count of modified rows; and control reads – the rows modified by the respective write. Thus, all the responses are checked for consistency.

As an assurance of correcteness of our approach, we compare the full database states, value by value, on both DBMSs after each experiment. We identified no inconsistencies in this way.

We instrumented the code with *configurable* fine-grained logging: by default, we log transaction-related data on the client side, and we optionally log at SQL operation level (on either, or both, client and DivSQL side). We used only the former in our comparison of DivSQL against single DBMSs (Sect. V.B and V.D) and non-diverse DBMS pairs (Sect. V.C), though the results remained unaffected by the chosen logging level.

For the performance evaluation, we use our own implementation of TPC-C (based on the SQL common to all chosen DBMSs), the industry-standard benchmark for OLTP. TPC-C is, thus, dominated by write transactions; only about 8% of the transactions are read-only. We have implemented TPC-C using either standard, or prepared, SQL statements. In our evaluation, we used the latter implementation due to its well-known performance. *Control SELECTs* were generated using prepared statements too, by parsing, and string manipulation of, the respective write operation *only once in the beginning of the experiment* – an additional performance benefit. We used TPC-C databases with 20 Warehouses. The minimum mean of TPC-C Think Times distribution were scaled down to the following values (in seconds): 1, 1, 0.8, 0.3 and 0.3 respectively for New-Order (NO), Payment (P), Delivery (D), Order-Status (OS), and Stock-Level (SL) transactions (TPC-C specifies 12, 12, 10, 5 and 5 seconds, respectively). We ran experiments with shorter, or no, Think Times, and the abort rate was, unsurprisingly, intolerable; this is due to a high contention exhibited in the workload, and especially via the Payment transaction.

Our focus is on the comparison of DivSQL performance, in both *Pess* and *Opt* regime, against the performances of single DBMSs ("1 DBMS"). When comparing to the DivSQL in the *Pess* regime 1 DBMS performance represents the lower bound for DivSQL. The 1 DBMS experiments use respective DBMS's JDBC driver, have a thread per connection to communicate with the server, and incur overhead executing through our middleware. The throughput results for single DBMS experiments were equivalent with, or without, the use of the `mutex`, an essential feature for our replication protocol to ensure consistency, but unneeded when running 1 DBMS experiments. Unlike in the *Pess* regime, and the same as in the *Opt* regime, we have *not* executed control SELECTs with 1 DBMS experiments – this caters for a fair comparison.

We also implemented TPC-C in a separate codebase, using prepared statements (as for DivSQL), to test single DBMSs *without any* of our replication code (*Single_Server* codebase). These results were similar to the ones obtained for 1 DBMS experiments where we used DivSQL codebase. This increases our confidence in the middleware being implemented efficiently.

Notably, the performance of individual DBMSs is, in general, going to differ, and potentially considerably so. The difference will affect how much one can limit the performance overhead of DivSQL, dictated by the slower DBMS, when using the *Pess* regime, and how much of improvement, if any, one can observe with the *Opt* regime. We used DBMSs without any optimisations, except the following FB parameters: `DefaultDbCachePages` and `FileSystemCacheSize`. They were set to optimise the use of memory for the architecture ("SuperServer") we used. We did this to bring closer the FB performance to that of the other DBMSs.

We used transactions per minute (*tpm*), adapted from the TPC-C's *tpmC* (number of new orders per minute under response time constraints), as the main performance metric, as well as mean transaction latency.

We used no vendor-specific SQL extensions from any of the DBMSs – achieving full vendor-agnostic SQL compatibility is outside of the scope of this work. In general, non-determinism in the results returned by a DBMS is possible, either since no specific ordering of results is used, or due to use of non-deterministic functions available in DBMSs (e.g., to generate timestamps). We preclude this by i) using `ORDER BY` clauses (some are, in any case, a part of the TPC-C implementation) in cases where more than one row/cell is returned (many SQL operations of the TPC-C implementation return a single row, however), and ii) generating timestamps on the client side, instead of invoking respective server-side functions.

NOWAIT was set on one of the DBMSs in a DivSQL pair. Each experiment comprises the same sequence of 100,000 transactions and was repeated 3 times. Also, we ran hundreds of experiments during the implementation – they are all consistent with the presented results. The VMs on which DBMSs ran were restarted, and the databases restored, between experiments.

B. *TPC-C Workload – Comparison with Single DBMSs*

The throughput (tpm) values for single servers (1Comm, 1FB, and 1PG) executing through our replication middleware code, and DivSQL (1Comm1PG and 1FB1PG, in *Opt* and *Pess* regimes) are given in Fig. 4 (1Comm1FB findings are consistent to the presented results and are omitted for paper length reasons).

Under 50 Clients load, 1Comm1PG *Pess*, made up of the two faster DBMSs, exhibits very modest performance overhead when compared to the fastest single DBMS (PG): just 6%. 1Comm1PG *Opt* is, however, faster than 1PG for the loads of 1, 5, 20 and 40 Clients – this confirms diversity potential for performance improvement. The opposite is true for 10, 30 and 50 Clients experiments. The differences are, however, not statistically significant in most cases: the p-value of Welch's two-sided t-test was greater than the confidence level of 0.05 indicating insufficient evidence to conclude a significant difference. Only for the following 2 comparisons the p-value was lower than the confidence level: 30 Clients, 1Comm1PG *Pess* vs 1PG – 0.009758; 50 Clients, 1Comm1PG *Opt* vs 1PG – 0.03873. Also, Coefficient of Variation (CV=std.dev./mean) values are all (significantly) less than 1 – this confirms low variability.

The transaction latencies are, on average, longer on DivSQL. For the 50 Clients experiment, the overhead of 1Comm1PG *Pess* compared to 1PG is about 20% when all transaction types are considered, and for 1Comm1PG *Opt* is 9%. Looking at average latency per transaction type, the overhead ranges 6%-46% for the *Pess*, and about 6%-30% for the *Opt*, regime. Notably, 1Comm1PG *Opt* exhibits shorter average latency for Delivery, a read-write transaction type – it is 3% faster on average than the 1PG, the faster of the two DBMSs. The average latency for all transactions, and for each of the 5 types, is faster on 1Comm1PG *Opt* than the individually slower DBMS – 1Comm. Similar results hold for the other DivSQL pairs.

The specific mix of SQL operations – high proportion of writes, interleaved with only short sequences of reads (usually only 1 or 2 SELECTs before a write is executed in a transaction) – precludes the *skip* feature being effective. Thus, the *Opt* regime is limited in performance gain it achieves.

The abort rate was no higher than 3.6% for single DBMS experiments, and no higher than 5.9% for DivRep (in both *Pess* and *Opt* regimes, including FB pairs) for the loads from 1 to 50 clients. Once we increased the load to 100 Clients, DivSQL with 1Comm1PG *Opt* exhibited average abort rate of 5.6%, while for 1Comm1PG *Pess* it was 8.4%, likely intolerable in most real-world applications (albeit TPC-C not specifying a threshold; and the rate recorded after we reduced Think Times by about an order of magnitude). Using FB, the slowest DBMS, in DivSQL increases the abort rate further. Thus, we did not perform 100 Clients experiment systematically with all configurations. Majority of aborts occurred for Payment transactions.

As expected (Sect. IV.A), NOWAIT exceptions were the most numerous ones raised: e.g., under 50 clients load, 33-69% more numerous than the other exceptions due to write-write conflicts, depending on the diverse pair and the regime used, and which DBMS in the pair had NOWAIT enabled.

The aborts are initiated by the client application, e.g., once notified of a concurrency conflict, as is indeed the standard practice, instead of initiating abort from DivSQL. This results in an additional network hop (between DivSQL and DBMSs) when compared with experiments using *Single_Server* codebase in which client communicates directly with a chosen DBMS.

Notably, when using *Single_Server* codebase, under demanding load with 100 Clients (1k transactions each), the mean throughput for 1PG was 4867.5 tpm. The mean throughput for 1Comm1PG DivSQL (when, clearly, we used our fully-fledged replication codebase) under the same load in *Opt* regime was 4431.8 tpm, and in *Pess* 4233.2 tpm. This is a modest overhead of 9%, and 13%, respectively, compared to 1PG experiment without use of any replication code. Similar results were obtained when comparing the DivSQL with 1Comm executed using *Single_Server* codebase (its throughput is similar to 1PG's). This comparison is an ultimate test of DivSQL performance, since it is compared to a "1 DBMS" configuration executing *without* any replication code, an approach not standardly used in evaluation of replicated databases.

We used our fine-grained logging on SQL operation level to estimate the overhead of Comparator function. We compared the transaction end timestamps with the respective end timestamps of the last SQL operation in the given transaction – the differences were small. This shows that, for the workloads we used, the Comparator function overhead is negligible.

We observed no result inconsistency raised by the Comparator during the experiments, unlike during our testing phase where we purposefully ran *Pess* regime experiments with the inconsistent initial states on the two DBMSs in DivSQL and observed the Comparator indeed reporting inconsistencies. Also, once we excluded the use of the `mutex`, allowing different sequences of begins and commits on the two DBMSs, the Comparator function detected the inconsistencies between the DBMSs' results and the transaction was aborted. Both offer an additional assurance DivRep is implemented correctly.

C. *TPC-C Workload – Comparison with Non-diverse Pairs*

Replicated DBMSs solutions use more resources than centralized counterparts. Thus, a useful performance evaluation of DivSQL is against replicated solutions that use DBMSs from the same vendor. We compared 1Comm1PG against the non-diverse pairs of both the faster (PG), and the slower (Comm), single server. We used TPC-C workload and the most demanding load of 50 Clients, each executing 2k transactions. In both cases non-diverse pairs (2PG, 2Comm) were deployed using our replication code, including the `mutex` to ensure data consistency. We enabled NOWAIT on only one of the 2 non-diverse DBMSs: we set server-wide `lock_timeout` parameter in one of the PGs, or the connection-level `Lock_Timeout` on a Comm. The average throughput of 2PGs in *Opt* regime was 2447.2 tpm – negligibly faster than 1Comm1PG *Opt* (2438 tpm) and 4% faster than 1Comm1PG *Pess* (2338.6 tpm). 2Comm *Opt* result (2441.6 tpm) was even closer to that of the diverse pair.

D. *Read-only Workload – Comparison with Single DBMSs*

We have derived a read-only workload based on the TPC-C, as in [10]: OS and SL transactions were executed at 50%. We executed experiments with 100 Clients (1k transactions each). The tpm values for individually faster single servers (1Comm

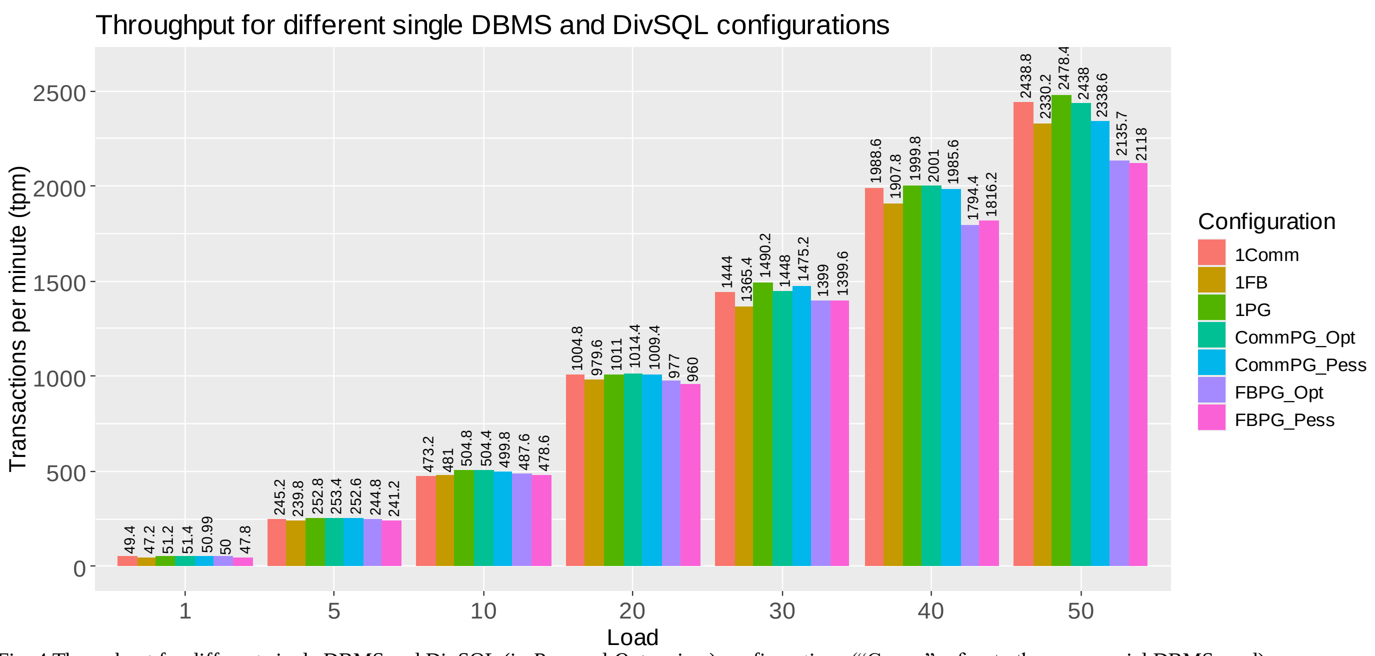

Fig. 4 Throughput for different single DBMS and DivSQL (in *Pess* and *Opt* regime) configurations (“Comm” refers to the commercial DBMS used).

and 1PG; we do not report 1FB results) executing through our replication middleware, and DivSQL (1Comm1PG, in *Opt* and *Pess* regime) are as follows: 7860.2 (1PG), 7853.4 (1Comm), 7843.2 (1Comm1PG *Opt*), 7805.8 (1Comm1PG *Pess*). The average transaction latencies, in msec, were as follows: 1PG ~ 8; 1Comm ~ 9; 1Comm1PG *Opt* ~ 10 and 1Comm1PG *Pess* ~ 15.

Since this is a read-only workload, we have not used the `mutex` for begins and commits synchronisation, in any of the 2 single DBMS, or the DivSQL, configurations. This is analogous to [10] (see Fig 2 in their paper). We kept the Think Times the same as in the TPC-C experiments (Sect. V.A).

The effectiveness of the *skip* feature, used with the *Opt* regime, is limited in the specific read-only workload we used. This is due to both OS and SL transactions being made up of a small number of SELECT operations: 3 and 2 respectively. The slower DBMS starts the execution of at least the first SQL operation, and likely all of them, in the transaction. This precludes the *skip* feature to be triggered and dynamically balance the load. This holds for the original TPC-C workload, too, but the effect is less pronounced since the two transaction types jointly contribute about 8% of the executions in TPC-C.

VI. Discussion

A. *Potential DivRep Optimisations*

Here we describe possible DivRep optimisations. They have not yet been implemented in our solution.

DivRep guarantees strict consistency among the replicas by imposing the same order of begins and commits on both. We can optimise the algorithm, when executing in either the *Pess* or the *Opt* regime of operation, in the following ways.

Firstly, we could relax the requirement that the order of begin operations is identical on the DBMSs. If no commit is executed in between a sequence of begins, different orderings of begins are allowed on different replicas. For example, assume three transactions T_0, T_1 and T_2 executing over two replicas R_x and R_y. A schedule of the transaction boundary operations on R_x is: c_0, $\mathbf{b_1}$, $\mathbf{b_2}$, c_1, c_2. An equivalent order of transaction boundaries: c_0, $\mathbf{b_2}$, $\mathbf{b_1}$, c_1, c_2 is allowed on R_y. Thus, a different sequence of begins is allowed to execute in parallel on the DBMSs, though any commit must be synchronised and it would be blocked until the begins are executed on both DBMSs. The equivalent histories of transaction boundaries are preserved in this way.

Also, a sequence of commits (*CommitSeq)* belonging to *non-conflicting* transactions (for which the respective writesets are disjoint) can be executed in different orders on the two replicas. Analogous to the preceding optimisation, the synchronisation “granularity” changes from a single commit operation to a sequence of commits. Ensuring that any begin is blocked until the *CommitSeq* members are executed ensures the same reads-from relations on both DBMSs.

Finally, it is unnecessary to synchronize the commits of read-only transactions. There are two possibilities. A transaction commit will not be synchronized with the commits and begins of concurrent transactions if the transaction has no writes (in [10] a transaction is assumed read-only until the first write), or DivRep checks that the transaction’s writeset is empty. The latter is similar to the functionality of other replication schemes [4], [26], [27] and can be performed using triggers or transactional logs, available in many DBMSs (MSSQL, Oracle, PostgreSQL, etc.).

B. *DivSQL Hybrid Approach*

DivSQL can be configured to run in different regimes of operation depending on the specific client requirements. The *Pess* regime offers improved fault–tolerance, by comparing the results of SQL operations from different DBMSs, while the complementary *Opt* regime is meant to deliver performance improvements by executing (most of) the read operations only on one DBMS.

These two regimes are not mutually exclusive – they can be combined into a configurable quality of service. By deploying *learning capabilities*, e.g., via Bayesian inference [28], DivSQL may process the individual SQL operations switching intelligently between the different regimes. The switch between the regimes will be driven by confidence gradually built by DivSQL that a particular type of operation is unlikely to cause a mismatch between the responses of the deployed diverse replicas. Before the predefined level of confidence is reached, whenever DivSQL receives an operation, it will process it under the *Pess* regime. As the number of instances of the same type of operation (i.e., the same operation with different parameter values) grows, and no mismatches are observed between the DBMSs' responses, so will the confidence that the particular type of operation is unlikely to lead to mismatches between the diverse replicas. Eventually, the predefined level of confidence will be reached, from which point DivSQL will execute the subsequent instances of the same operation type under the *Opt* regime. A mismatch between the DBMS responses during the learning period will either lead to DivSQL processing all future instances of the operation under the *Pess* regime or require a significantly greater number of identical responses to reach the predefined level of confidence.

When multiple applications execute against the same DivSQL, and some of them are not subjected to runtime assessment using the learning capabilities, a special care must be taken. This is because data inconsistencies might be introduced with the applications that do not use the *hybrid* approach, and determining the switching point between the two regimes, for applications that seek improved dependability, could be invalidated. To prevent this, DivSQL can initiate *periodic* consistency checks, after the switch between the regimes had occurred and executing in the *Opt* regime is taking place. This will also help reveal the cases where, despite the initial execution in the *Pess* regime, an inconsistency is triggered after the switching point.

Learning capabilities could also be used for determining which DBMS is faster for a particular type of read. Consequently, the read would be executed only on the faster DBMS. The load of the reads would be divided between the replicas once DivSQL learns which is the fastest for all the reads. This could be more efficient than the *skip* feature because no read operation would be executed on more than one replica. Some feedback data would need to be provided to the load balancing technique so that changes in latencies, e.g., a faster DBMS starts to work more slowly under different workload, are detected. The technique resembles ROWA(A), but is more flexible, since DivSQL would have alternatives in deciding on which DBMS to execute a particular read operation e.g., using additional load balancing information a read could execute on a slower DBMS in the cases when it is being subjected to a lighter load.

VII. Conclusions and Future Work

We presented design and implementation of DivSQL, a novel middleware-based database replication solution, and its replica control protocol (DivRep), for SI-enabled DBMSs. DivSQL assumes realistic IRFM failure model, adopted based on the convincing experimental evidence [1, 2]. DivSQL ensures stringent consistency for replicated SI-enabled DBMSs: *strict 1-copy-SI*, based on 1-copy-SI [4], and *Conventional Snapshot Isolation* [5], and we provide respective proofs.

A comprehensive experimental evaluation of DivSQL using 3 DBMSs is provided. When the performances of the individual servers are similar, DivSQL in *Pess* regime exhibits overhead of only 6% compared to the faster single DBMS. This is a tolerable overhead given the dependability assurance DivSQL offers through deployment of diverse DBMSs.

When DivSQL *Opt* is used, it performs better on average for some loads than the faster of the two diverse DBMSs. To further improve the performance of DivSQL *Opt*, we plan to leverage the inherent diversity in performance of individual DBMSs for intelligent load balancing (Sect. VI.B). We will investigate criteria for choosing diverse DBMSs, i.e., choose the ones known to use different approaches to complex operations and exhibit systematic differences.

Deferred update technique (e.g. [29], [30]) has been used for performance improvement, and we plan to explore it for DivSQL. A challenge is a suitable comparison of changes made by *diverse* replicas, and doing it in an efficient way since the overhead of the writeset comparison in the certification phase in the end of transaction might be significant.

To improve scalability of DivSQL, we will explore the possibilities of connecting multiple DivSQL nodes using a complementary replication protocol, that achieves high scalability, into a "fusion" solution.

We plan to evaluate DivSQL performance using other workloads, and under different isolation levels, e.g., read-committed – the default in many DBMSs, as well as analyse whole (transaction and operation) latency distributions, rather than only averages, which is often missing in the studies of this kind. Also, we plan to compare DivSQL against existing fault-tolerant SI-based database replication solutions, e.g., Byzantium [10], and Snapshot Epoch Scheduling [9].

We plan to evaluate diversity effectiveness using the measurements on the SQL operation level, e.g., show proportion of faster responses coming from either of the two DBMSs in DivSQL, and absolute and relative gains achieved in this way.

We plan to conduct a performability-style analysis, and evaluate the loss attributed to "incorrect results" failures using fault-injection.

OLTP applications are increasingly executed against memory-resident databases that improve performance of transaction processing, pointed out back in 2008 by Stonebraker et al. [31], e.g., in-memory tables have been available in MSSQL since v14. We plan to evaluate DivSQL using those DBMSs.

Acknowledgement

We thank the anonymous reviewers and our shepherd, Fernando Pedone, for providing valuable feedback on earlier versions of this paper. This work was partially supported by the EPSRC DIDERO-PC project EP/J022128/1.

REFERENCES

[1] Gashi, I., P. Popov, and L. Strigini, *Fault tolerance via diversity for off-the-shelf products: a study with SQL database servers.* IEEE Transactions on Dependable and Secure Computing, 2007. **4**(4): p. 280-294

[2] Vandiver, B., et al., *Tolerating byzantine faults in transaction processing systems using commit barrier scheduling*, in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*. 2007, ACM: Stevenson, Washington, USA. p. 59-72.

[3] Bernstein, A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. 1987, Reading, Mass.: Addison-Wesley.

[4] Lin, Y., et al. *Middleware Based Data Replication Providing Snapshot Isolation*. in *ACM SIGMOD International Conference on Management of Data*. 2005. Baltimore, Maryland: ACM Press.

[5] Elnikety, S., W. Zwaenepoel, and F. Pedone. *Database Replication Using Generalized Snapshot Isolation*. in *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. 2005. IEEE Computer Society.

[6] Berenson, H., et al. *A Critique of ANSI SQL Isolation Levels*. in *SIGMOD International Conference on Management of Data*. 1995. San Jose, California, United States: ACM Press New York, NY, USA.

[7] Gray, J., et al. *The Dangers of Replication and a solution*. in *ACM SIGMOD International Conference on Management of Data*. 1996. Montreal, Canada: SIGMOD.

[8] Wiesmann, M., F. Pedone, and A. Schiper. *Database Replication Techniques: a Three Parameter Classification*. in *19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*. 2000. Nurnberg, Germany: IEEE.

[9] Vandiver, B., *Detecting and Tolerating Byzantine Faults in Database Systems*, in *Programming Methodology Group*. 2008, Massachusetts Institute of Technology: Boston. p. 176.

[10] Garcia, R., R. Rodrigues, and N. Preguica, *Efficient middleware for byzantine fault tolerant database replication*, in *Proceedings of the sixth conference on Computer systems (EuroSys '11)*. 2011, ACM: Salzburg, Austria. p. 107-122.

[11] Pedone, F. and N. Schiper, *Byzantine fault-tolerant deferred update replication.* Journal of the Brazilian Computer Society, 2012. **18**(1): p. 3-18.

[12] Molina, H.G., F. Pittelli, and S. Davidson, *Applications of Byzantine agreement in database systems.* ACM Trans. Database Syst., 1986. **11**(1): p. 27-47.

[13] Littlewood, B., P. Popov, and L. Strigini, *Modelling Software Design Diversity - A Review.* ACM Computing Surveys, 2001. **33**(2): p. 177-208.

[14] Strigini, L., *Fault Tolerance Against Design Faults*, in *Dependable Computing Systems: Paradigms, Performance Issues, and Applications*, H. Diab and A. Zomaya, Editors. 2005, John Wiley & Sons. p. 213 - 241.

[15] Gashi, I. and P. Popov. *Rephrasing Rules for Off-the-Shelf SQL Database Servers*. in *Sixth European Dependable Computing Conference (EDCC '06).* 2006.

[16] Wiesmann, M., et al. *Understanding replication in databases and distributed systems*. in *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*. 2000. Taipei, Taiwan: IEEE Computer Society Los Alamitos.

[17] Fekete, A., et al., *Making Snapshot Isolation Serializable.* ACM Transactions on Database Systems (TODS), 2005. **30**(2): p. 492 - 528.

[18] Fekete, A., E. O'Neil, and P. O'Neil, *A read-only transaction anomaly under snapshot isolation.* ACM SIGMOD Record, 2004. **33**(3): p. 12-14.

[19] Rigger, M. and Z. Su, *Finding bugs in database systems via query partitioning.* Proc. ACM Program. Lang., 2020. **4**(OOPSLA): p. Article 211.

[20] Gray, J. and L. Lamport, *Consensus on transaction commit.* ACM Trans. Database Syst., 2006. **31**(1): p. 133–160.

[21] Behm, A., S. Rielau, and R. Swagerman. *Returning Modified Rows – SELECT Statements with Side Effects*. in *Proceedings of the Thirtieth International Conference on Very Large Data Bases*. 2004. Toronto, Canada: Morgan Kaufmann.

[22] Skeen, D. *Nonblocking commit protocols*. in *Proceedings of ACM SIGMOD International conference on management of data*. 1981. Ann Arbor, Michigan: ACM Press, New York, NY, USA.

[23] Budhiraja, N., et al., *The primary-backup approach*, in *Distributed systems (2nd Ed.)*. 1993, ACM Press/Addison-Wesley Publishing Co. p. 199–216.

[24] Raz, Y., *Serializability by commitment ordering.* Information Processing Letters, 1994. **51**(5): p. 257-264.

[25] Fischer, M.J., N.A. Lynch, and M.S. Paterson, *Impossibility of distributed consensus with one faulty process.* J. ACM, 1985. **32**(2): p. 374–382.

[26] Kemme, B. and S. Wu. *Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation*. in *International Conference on Data Engineering*. 2005. Tokyo, Japan: IEEE Computer Society.

[27] Patino-Martinez, M., et al., *MIDDLE-R: Consistent database replication at the middleware level.* ACM Transactions on Computer Systems (TOCS), 2005. **23**(4): p. 375-423.

[28] Gorbenko, A., et al., *Dependable composite web services with components upgraded online*, in *Architecting Dependable Systems III*. 2005, Springer-Verlag. p. 92–121.

[29] Pedone, F., R. Guerraoui, and A. Schiper, *The Database State Machine Approach.* Distributed and Parallel Databases, 2003. **14**(1): p. 71-98.

[30] Kemme, B. and G. Alonso, *A new approach to developing and implementing eager database replication protocols.* ACM Transactions on Database Systems (TODS) 2000. **25**(3): p. 333 - 379.

[31] Harizopoulos, S., et al., *OLTP through the looking glass, and what we found there*, in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 2008, Association for Computing Machinery: Vancouver, Canada. p. 981–992.

Appendix A – DivRep Correctness

This section contains the proof of correctness of DivRep when either of the following two criteria for consistency of replicated databases are considered: 1-copy-SI [4] and Generalized/Conventional Snapshot Isolation (GSI/CSI) [5]. We include the relevant parts of each criterion, in Sect. "DivRep Correctness Based on 1-copy-SI" and Sect. "DivRep Correctness Based on CSI/GSI", respectively, to aid reader's comprehension. The correctness of DivRep is not limited to the use of two replicas in DivSQL – it is ensured for an arbitrary number of replicas.

A. *DivRep Correctness Based on 1-copy-SI*

Lin et al. [4] defined criteria for correctness of replicated databases when each of the underlying replicas guarantees SI. The correctness criterion, referred to as *1-copy snapshot isolation* (1-copy-SI), guarantees that an execution of transactions over a set of replicas produces a global schedule that is equivalent to a schedule produced by a centralised database system which offers snapshot isolation. The authors provide the following three definitions to formalise 1-copy-SI correctness:

Definition 1 (SI-Schedule). *Let T be a set of committed transactions, where each transaction T_i is defined by its readset RS_i and writeset WS_i. An SI-schedule S over T is a sequence of operations $o \in \{b, c\}$. Let $(o_i < o_j)$ denote that o_i occurs before o_j in S. S has the following properties.*

i. *For each $T_i \in T$: $(b_i < c_i) \in S$.*

ii. *If $(b_i < c_j < c_i) \in S$, then $WS_i \cap WS_j = \{\}$.*

The read and write operations are excluded from *Definition 1* because the transaction boundary operations, begin (b) and commit (c), implicitly determine the logical time of their executions: a begin of transaction T_i indicates when its reads have taken place and similarly the commit of T_i indicates when the write operations take effect. This reasoning is based on the characteristics of SI (see Sect. III).

Definition 2 (SI-Equivalence). *Let S^1 and S^2 be two SI-schedules over the same set of transactions T. S^1 and S^2 are SI-equivalent if for **any** two transactions T_i, $T_j \in T$ the following holds:*

i. *if $WS_i \cap WS_j \neq \{\} : (c_i < c_j) \in S^1 \Leftrightarrow (c_i < c_j) \in S^2$.*

ii. *if $WS_i \cap RS_j \neq \{\} : (c_i < b_j) \in S^1 \Leftrightarrow (c_i < b_j) \in S^2$.*

Definition 2 is based on the equivalence definitions as specified for the non-replicated database systems using serializability theory. Condition *i.* ensures that the order of committed transactions with overlapping writesets is the same in both schedules. Thus, the final writes (a write performed by a committed transaction after which no other committed transaction modified the same data item) are the same in the two schedules and each *prefix* of the partial order of committed transactions in both schedules is an SI schedule. Condition *ii.* ensures that if in one schedule a transaction, T_j, reads data modified by a committed transaction, T_i, the same will be true for the other schedule – the begin of T_j will follow the commit of T_i.

In order to define 1-copy-SI criterion the authors of [4] assume the following:

- Each replica produces SI schedules.
- Replication is based on ROWA approach: each transaction is executed on a local replica and only its writes are propagated to the remaining ones. To formalise the ROWA approach the authors use a mapper function *rmap*. The input to the function is a set of transactions $\mathcal{T}$ and a set of replicas $\mathcal{R}$. Each update transaction is transformed into a set of transactions $\{T_i^k | R^k \in \mathcal{R}\}$, one for each replica. Only one of these transformed transactions contains both, the read and the write set of the original transaction - this is the *local* transaction. The rest of the transactions are remote and consist of only the writeset of the transaction. Every read transaction, on the other hand, has a single transformation into a local transaction.

Definition 3 (1-Copy-SI). *Let $\mathcal{R}$ be a set of replicas following ROWA approach. Let $\mathcal{T}$ be a set of submitted transactions for which $T_i \in \mathcal{T}$ committed at its local site. Let S^k be the SI-schedule over the set of committed transactions T^k at replica $R^k \in R$.*

Then $\mathcal{R}$ ensures 1-copy-SI if the following is true:

i. *There is ROWA mapper function,* rmap*, such that $\bigcup_k \mathcal{T}^k =$* rmap $(\mathcal{T}, \mathcal{R})$

ii. *There is an SI-schedule S over T such that for each S^k and T_i^k, $T_j^k \in \mathcal{T}^k$ being transformations of T_i, $T_j \in \mathcal{T}$:*

a. *if $WS_i^k \cap WS_j^k \neq \{\} : (c_i^k < c_j^k) \in S^k \Leftrightarrow (c_i < c_j) \in S$,*

b. *if $WS_i^k \cap RS_j^k \neq \{\} : (c_i^k < b_j^k) \in S^k \Leftrightarrow (c_i < b_j) \in S$.*

From the condition *i.*, we infer an existence of an *rmap* function that maps committed transactions as a subset of the set of submitted ones. Condition *ii.* ensures equivalence between a schedule produced by a replica, S^k, and the global schedule, *S*, over the set of all transactions $\mathcal{T}$. Due to the use of ROWA approach, the definition of equivalence as stated in *Definition 2* has to be modified. The condition *i.* from the *Definition 2* holds between every S^k and *S* for all committed transactions, because the writes are executed on all replicas. But, the reads-from relation of a schedule S^k is the same as in *S* (condition *ii.* from *Definition 2*) for only the subset of the readsets obtained at the replica R^k. There are two consequences of the 1-copy-SI definition:

- The position of the begin operations of remote transactions is arbitrary since they do not include read operations.
- The position of the commits of the read-only transactions is arbitrary since they do not include any write operations.

The following text until the end of the subsection gives proof of DivRep with respect to 1-copy-SI. The following definition, *Strict 1-copy-SI,* is based on 1-copy-SI [4] – ***Definition 3***. We use the definition of SI-Equivalence ***Definition 2*** for formalising *Strict 1-copy-SI*. The difference is in removing the ROWA restriction, where reads are executed only at a local site. We are interested only in the set of committed transactions [3].

Definition 4 (Strict 1-copy-SI). *Let $\mathcal{S}$ be a set of schedules, $\mathcal{R}$ a set of replicas and $\mathcal{T}$ a set of submitted transactions. Let S^k be an SI-schedule over the set of committed transactions on replica R^k. We say that $\mathcal{R}$ provides Strict 1-copy-SI if all schedules from $\mathcal{S}$ are pairwise SI-equivalent.*

Assumption 1: The underlying replicas provide SI, which is implemented using Strict 2-Phase Locking and multiversioning.

Assumption 2: Once all the write locks are acquired, all write (and read) operations are executed, and a replica has voted for commit, the commit is guaranteed to succeed on the given replica[3].

Proposition 1: All replicas commit the same set of transactions.

Proof: After a transaction is submitted, it commits either on all replicas or at none. This follows from the fact that a transaction termination is performed using an atomic commitment protocol, AC-DR (see II.C. and II.D. in Fig. 2), where all replicas agree on an outcome, commit or abort, i.e., uniform agreement is guaranteed (*Assumption 2* ensures commits are successful on all replicas that voted for commit).

Q.E.D.

Theorem 1. DivRep guarantees *Strict 1-copy-SI*.

Proof: Assume it does not. Then there exists a pair of schedules $(S^1, S^2) \in \mathcal{S}$ and a pair of transactions $(T_i, T_j) \in \mathcal{T}$ for which the following holds:

i. $WS_i \cap WS_j \neq \{\} : (c_i < c_j) \in S^1$ ***and*** $(c_j < c_i) \in S^2$.

or

ii. $WS_i \cap RS_j \neq \{\} : (c_i < b_j) \in S^1$ ***and*** $(b_j < c_i) \in S^2$.

Both *i.* and *ii.* are impossible because *Proposition 1.* holds and the transaction boundary operations are executed atomically in DivRep – the same order of transaction boundary operations is executed on the replicas (see II.A.2-4 for begins, and II.D.1.a.i-iv for commits, in Fig. 2).

Q.E.D.

The above proof holds for both *Pess* and *Opt* regime of DivRep. In the *Opt* regime, some of the reads might be *skipped*, but the uniform agreement and the identical order of transaction boundary operations is maintained.

B. DivRep Correctness Based on CSI/GSI

Elnikety et al. [5] defined Generalised Snapshot Isolation (GSI) – a correctness criterion for replicated databases that offer snapshot isolation. GSI is an extension to the snapshot isolation (SI) as found in centralized databases. The authors formalize the SI used in centralised databases as Conventional Snapshot Isolation (CSI), in which the transaction reads the *latest* snapshot in the whole (replicated/centralised) database, not only the latest snapshot available locally on a replica (Prefix-Consistent Snapshot Isolation (PCSI)) – this is the strictest form of GSI. DivRep ensures CSI, a special, most demanding, case of GSI.

To model the timing relationships between transactions the following definitions in a transaction, T_i, are given:

- *snapshot(T_i)* – the time when T_i's snapshot is taken.
- *start(T_i)* – the time of the first operation of T_i.
- *commit(T_i)* – the time of commit of T_i.
- *abort(T_i)* – the time of abort of T_i.

In addition, they showed that serializability can be guaranteed under GSI by ensuring that either a static property, which can be checked by examining the transactional profile, or a dynamic one, which checks the intersection between the readsets and writesets of overlapping transactions, is satisfied.

In CSI each transaction sees the last snapshot regarding its starting time, i.e., *snapshot(Ti) = start(Ti)*.The definitions of GSI and CSI, and the corresponding definitions of *impacting* transactions, are as follows:

Generalised Snapshot Isolation (GSI) Definition:

- *G1. (GSI Read Rule)*

  $\forall\ T_i, X_j$ *such that* $R_i(X_j) \in h$:

  1. $W_j(X_j) \in h$ ***and*** $C_j \in h$;
  2. $commit(T_j) < snapshot(T_i)$;
  3. $\forall\ T_k$ *such that* $W_k(X_k), C_k \in h$:

  $commit(T_k) < commit(T_j)$ ***or*** $snapshot(T_i) < commit(T_k)$;

- *G2. (GSI Commit Rule)*

  $\forall\ T_i, T_j$ *such that* $C_i, C_j \in h : \neg\ (T_j$ *impacts* $T_i)$;

Definition of Impacting Transactions for GSI:

- *T_i impacts T_j* ***iff:***

  $snapshot(T_i) < commit(T_j) < commit(T_i)$ ***and***

  $writeset(T_i) \cap writeset(T_j) \neq \{\}$

Conventional Snapshot Isolation (CSI) Definition:

- *C1. (CSI Read Rule)*

  $\forall\ T_i, X_j$ *such that* $R_i(X_j) \in h$:

  1. $W_j(X_j) \in h$ ***and*** $C_j \in h$;

  2. $commit(T_j) < snapshot(T_i)$;

[3] A commit execution on a SI DBMS can, in principle, fail and throw an exception for reasons different than SI write-write conflicts or "incorrect results", i.e., the failure model DivRep guards against (IRFM). Such a subtle Byzantine failure is outside the scope of DivRep: a replica that initially "voted" for commit during the certification subsequently aborts the transaction. This is an unlikely event, confirmed by our extensive empirical evidence: after running, literally, billions of transactions under different configurations and workloads, our commit exception handler, which merely exits the application/middleware, has caught **no** such exception. IRFM – DivRep's realistic failure model – was chosen based on comprehensive experimental evidence of the faults that occur in practice.

3. $\forall$ T_k *such that* $W_k(X_k)$, $C_k \in h$:

$commit(T_k) < commit(T_j)$ **or** $start(T_i) < commit(T_k)$;

- *C2. (CSI Commit Rule)*

  $\forall$ T_i, T_j *such that* C_i, $C_j \in h$: $\neg$ (T_j *impacts* T_i)

Definition of Impacting Transactions for CSI:

- T_i *impacts* T_j ***iff:***

  $start(T_i) < commit(T_j) < commit(T_i)$ ***and***

  $writeset(T_i) \cap writeset(T_j) \neq \{\}$

CSI states that the *last* snapshot, committed on any of the database replicas, in respect to the transaction start time, is available. CSI is a special case of GSI as the latter does not specify which database snapshot should a transaction observe, i.e., $snapshot(T_i) = start(T_i)$ in CSI. The difference between GSI and CSI could be illustrated with the following example. Let history $h = W_i(X_i), C_i, W_j(X_j), C_j, R_k(X_i), W_k(Y_k), C_k$. The history is not permitted by CSI because T_k reads an "old" snapshot, $snapshot(T_i)$, instead of the *last* one, $snapshot(T_j)$. However, h is a GSI history since $snapshot(T_k) = commit(T_i)$ is allowed.

The following text until the end of the subsection provides proof of DivRep with respect to CSI.

Assumption 1: Underlying replicas ensure CSI
Theorem 1: DivRep ensures CSI

- DivRep ensures conditions C1.1 and C1.2 of CSI (see above) because the underlying replicas are assumed to guarantee CSI where only updates of committed transactions are visible i.e., no dirty reads are allowed.
- DivRep guarantees condition C1.3 of CSI – every transaction observes the *last* committed snapshot on any replica.

  Assume C1.3 was not ensured. Then it is possible for a transaction to read an "old" snapshot (we denote this property $\neg$ C1.3):

  $\exists$ T_i, T_k, X_j such that $R_i(X_j) \in h$ and $W_k(X_k)$, $C_k \in h$: $commit(T_j) < commit(T_k)$ **and** $commit(T_k) < start(T_i)$;

  - $\neg$ C1.3 is possible **only** if a replica produces such a schedule since in DivRep every transaction starts atomically on all replicas using the `mutex` and an identical order of begins and commits is ensured (see Fig. 2: II.A.2-4 for begins, II.D.1.a.i-iv for commits).
  - However $\neg$ C1.3 contradicts *Assumption 1*.
- DivRep enforces C2 (CSI Commit Rule)

  Assume it does not. Then it is true that impacting transactions are allowed (we denote the property $\neg$ C2):

  $\exists$ T_i, T_j such that C_i, $C_j \in h$:
  $start(T_i) < commit(T_j) < commit(T_i)$ **and**
  $writeset(T_i) \cap writeset(T_j) \neq \{\}$

  This is impossible since replicas provide snapshot isolation and a transaction will be aborted by DivRep if an "impact" i.e. *write-write* conflict, has been detected on any of the replicas (see I.B.2.d in Fig. 3); the abort is propagated to the other replica(s).
  Q.E.D.