Towards Auto-Scaling Existing Transactional Databases with Strong Consistency

Michael A. Georgiou Aristodemos Paphitis Michael Sirivianos Herodotos Herodotou
Cyprus University of Technology
{mica.georgiou, am.paphitis}@edu.cut.ac.cy {michael.sirivianos, herodotos.herodotou}@cut.ac.cy

***Abstract*—Existing relational database systems often suffer from rapid increases or significant variability of transactional workloads but lack support for scalability or elasticity. Database replication has been employed to scale workload performance but past approaches make various performance versus consistency tradeoffs and typically lack the mechanisms and policies for dynamically adding and removing replicas. This paper presents Hihooi, a replication-based middleware system that is able to achieve scalability, strong consistency, and elasticity for existing transactional databases. These features are enabled by (i) a novel replication algorithm for propagating database modifications asynchronously and consistently to all replicas at high speeds, and (ii) a new routing algorithm for directing incoming transactions to consistent replicas. Our experimental evaluation validates the high scalability and elasticity benefits offered by Hihooi, which form the key ingredients towards a truly auto-scaling system.**

***Index Terms*—database replication, scalability, end elasticity**

I. Introduction

The proliferation of internet-based services and applications has led to both a rapid growth and high variability of transactional workloads, which can negatively effect the performance of the underlying database system. However, most existing database systems do not offer any features to automatically support *workload scalability* (i.e., the ability to handle increasing workload demands) or *elasticity* (i.e., the ability to handle variations in those workloads). The main notable exceptions are some *Database-as-a-Service* (*DBaaS*) offerings (e.g., Amazon RDS, Azure SQL DB) that support some form of scalability and elasticity in a pay-as-you-go model. Yet, their adoption has been slow due to high costs for rewriting legacy applications and retraining employees, as well as privacy and security concerns [1]. Our vision is to offer automatic scalability and elasticity to existing transactional databases without application or database modifications.

Database replication has been successfully used for increasing performance and availability of databases by fully replicating data across multiple database nodes [2]. In its *master-slave* variant, one primary copy handles all write transactions while the other replicas process only read transactions [3], [4]. As long as the master node can handle the write workload, the system can scale linearly with the addition of more slave nodes [2]; making this approach an ideal candidate towards an auto-scaling system. The biggest challenges here are in (i) handling the trade-off between performance and consistency of the overall system, and (ii) dynamically adding and removing replicas while the system is running.

Existing replication-based approaches, however, have several limitations. Open-source solutions for replication are database-specific. MySQL Cluster [5] uses a synchronous replication mechanism that limits scalability. Postgres-R [6] integrates replica control into the kernel of PostgreSQL and utilizes special multicast primitives to propagate low-level write operations to the replicas. Middle-R [4], the middleware extension of Postgres-R, also requires database engine modifications for extracting and applying tuple-based modifications. Finally, Ganymed [3] is a master-slave replication middleware that applies all changes serially at the replicas, which causes replicas to fall behind the primary copy.

This paper presents *Hihooi*, *a replication-based master-slave middleware system that is able to achieve workload scalability, strong consistency, and elasticity for transactional databases*. An existing database can readily become the master in a Hihooi deployment. Replication is then used to increase throughput (via increasing the processing capacity of the system) and decrease latency (via spreading the load across the nodes). As a middleware system, Hihooi sits between the database engines and the clients, offering a single database view and masking the complexity of the underlying replication. Neither the database engines nor the clients need to be modified as long as the popular ODBC/JDBC drivers are used. Scalability and consistency are enabled by (i) a novel replication algorithm for propagating write transactions *asynchronously* and applying them *in parallel* (and consistently) to all replicas, and (ii) a new routing algorithm for *load balancing* read transactions to consistent replicas. Finally, elasticity is achieved by supporting an easy and quick way to add and remove replicas from the cluster.

Potential alternatives: Another database replication approach is called *multi-master*, in which all replicas serve both read and write transactions. However, explicit synchronization mechanisms are needed in order to agree to a serializable execution order of transactions, so that each replica executes them in the same order [6]–[8]. Concurrent transactions might conflict, leading to aborts and thus limiting the system's scalability [9]. *Data partitioning* (or *sharding*) is another scale-out approach, based on which the database data is partitioned and spread across all nodes [10]. While this approach does improve scalability (up to a point due to distributed transactions), it also requires expensive data migration and extensive manual physical design tuning for partitioning the data effectively

[11]. More recently, a new class of systems has arisen, called *NewSQL*, that offers scalable performance while still maintaining the ACID guarantees of a traditional database system [12]. NewSQL systems, however, are often highly optimized for a narrow set of use cases (e.g., MemSQL [13] is tuned for clustered analytics) and require other compromises related to language support or transaction and workload handling capabilities (e.g., in VoltDB [14], the unit of transaction is a Java stored procedure).

Contributions: In summary, the paper's contributions are:

1) A **database replication middleware architecture** for achieving workload scalability and consistency (Section II);
2) A **statement replication algorithm** for applying writes in parallel while ensuring consistent replicas (Section III);
3) A transaction-level **routing algorithm** for executing read transactions consistently and efficiently (Section IV);
4) An **implementation and evaluation** showcasing the scalability and elasticity attainable with Hihooi (Section V).

II. System Architecture

Hihooi is a *master-slave replication-based middleware* positioned between the applications and the database engines. The architecture and flow of transactions in Hihooi is shown in Figure 1. Existing applications can access Hihooi and the underlying databases using the custom **Hihooi JDBC/ODBC Drivers**, without requiring any code changes. Internally, Hihooi uses JDBC drivers for interacting with the underlying database engines in order to execute the queries and to manage replication behind the scenes. Hence, Hihooi is not coupled to the database engines, thus supporting multiple vendors.

All query requests are intercepted by the **Transaction Manager** and categorized into *write transactions* when at least one of the containing queries modifies the database (e.g., INSERT, UPDATE, DELETE SQL statements) and *read transactions* otherwise. The write transactions are directed to the master node, denoted as **Primary DB**, while the read transactions are load balanced to consistent slave nodes, denoted as **Extension DBs**. As long as the Primary DB can handle all writes and the system propagates the writes to the Extension DBs efficiently, the system can scale linearly by adding more Extension DBs. Once a write transaction completes on the Primary DB (either via `commit` or `rollback`), the transaction's statements are pushed into the **Transactions Buffer**, which is distributedly stored in memory using Memcached [15]. The write statements are then applied to the Extension DBs *asynchronously* in order to avoid delaying the write transactions executing at the Primary DB. Hence, the Transactions Buffer acts as a highly available propagation medium for all database modifications.

An **Extractor** service running on each Extension DB node is responsible for fetching the new write transactions from the Transactions Buffer and applying them to the local database in order to keep it consistent with the rest of the system. The Extractors implement a novel algorithm (discussed in Section III) for executing the transactions in parallel, while respecting the order imposed by the transaction commit timestamps on the Primary DB. In this manner, the Extension DB reflects

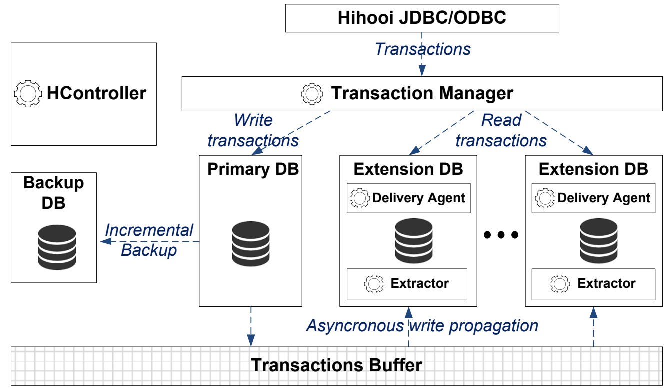


Fig. 1: Hihooi Architecture

a transaction-consistent snapshot of the data at the Primary DB; that is, it reflects all data modifications of transactions executed at the Primary DB in the same order of execution.

The asynchronous propagation of write transactions means that Extension DBs may not always be up-to-date with the Primary DB. Hence, Hihooi needs an efficient approach for determining which Extension DBs are consistent with which incoming read queries. The proposed solution consists of three parts. First, Hihooi extracts the tables, columns, or rows that are potentially modified or accessed by each incoming query. Second, Hihooi keeps track of the completed transactions that have been applied on each of the Extension DBs and, hence, recognizes which tables, columns, or rows are up-to-date on each of the Extension DBs. Finally, Hihooi employs a novel lightweight algorithm for checking which read queries are safe (from a consistency point of view) to execute on which Extension DBs (discussed in Section IV). If no consistent Extension DB is found, then Hihooi routes the request to the Primary DB, which is always consistent. A **Delivery Agent** is responsible for executing the read-only queries routed to the local Extension DB and delivering the results set incrementally to the client when requested, to avoid creating an execution bottleneck at the Transaction Manager.

The **Backup DB** represents a consistent checkpoint of the Primary DB at some point in time. The Backup DB is used to initialize a new Extension DB rather than the Primary DB to avoid any overheads imposed to the Primary DB. Next, the Extractor applies all appropriate write transactions from the Transactions Buffer and at the same time notifies the Transaction Manager that it can start serving read queries. Finally, the **HController** maintains various system statistics and coordinates all system management operations, such as adding and removing replicas. The HController will host *pluggable policies* for automatically deciding when to add or remove Extension DBs, which is our immediate future work.

III. Replication Management

Hihooi intercepts and redirects all incoming write transactions to the Primary DB. As soon as a transaction completes on the Primary DB, it must be propagated and executed on all

TABLE I: Example write transactions on tables $R(\underline{A1}, A2, A3, A4)$ and $S(\underline{B1}, B2, B3, B4, B5)$ along with corresponding write sets, read sets, affecting classes, and transaction state identifiers (TSIDs)

TX	SQL Statement	Write Sets			Read Sets			Affecting Class	TSID
		Table	Column	Row	Table	Column	Row		
W_1	UPDATE R SET A2 = ?, A3 = ? WHERE A1 = 100	R	$A2, A3$	$A1 = 100$	R	$A1$	$A1 = 100$	RAS	11
W_2	UPDATE S SET B2 = ? WHERE B5 > ?	S	$B2$		S	$B5$		CAS	12
W_3	UPDATE R SET A3 = ?, A4 = ? WHERE A2 < ?	R	$A3, A4$		R	$A2$		CAS	13
W_4	DELETE FROM R WHERE A1 = 120	R	$*$	$A1 = 120$	R	$A1$	$A1 = 120$	RAS	14
W_5	UPDATE S SET B4 = ? WHERE B5 < ?	S	$B4$		S	$B5$		CAS	15

TABLE II: Example read transactions on tables $R(\underline{A1}, A2, A3, A4)$ and $S(\underline{B1}, B2, B3, B4, B5)$ along with the corresponding read sets, affecting classes, and consistent transaction state identifiers (TSIDs)

TX	SQL Statement	Read Sets			Affecting Class	Consistent TSID
		Table	Column	Row		
R_1	SELECT * FROM R WHERE A2 > ?	R	$*$		TAS	14
R_2	SELECT A3, A4 FROM R WHERE A1 = 100	R	$A1, A3, A4$	$A1 = 100$	RAS	13
R_3	SELECT B2, B3 FROM S WHERE B5 < ?	S	$B2, B3, B5$		CAS	12
R_4	SELECT A1, B2, B3 FROM R JOIN S ON A1 = B2	R, S	$A1, B2, B3$		CAS	14

Extension DBs, while preserving the completion order from the Primary DB. Before explaining our replication procedure in Section III-B, we first introduce the notion of *transaction read/write sets* in Section III-A.

A. Transaction Read/Write Sets

Transactions are naturally divided into *single* and *multi-statement*, depending on the number of SQL statements included in the transaction. Each transaction T will read and/or modify some tables in a database, defined as the *Table Read Set* and the *Table Write Set* of T, respectively. For example, transactions W_1 and W_2 shown in Table I modify the respective tables R and S; these tables form the corresponding table write sets. Read/write sets allow us to reason about which transactions affect which tables. Thus, they allow us to effectively decide when to parallelize the execution of transactions on the Extension DBs (discussed in Section III-B) and how to route read transactions efficiently (see Section IV). For instance, W_1 and W_2 can be executed in parallel on the Extension DBs since they modify two different tables, regardless of their commit order on the Primary DB.

Operating with table read/write sets constitutes a coarse-grained mechanism for reasoning about conflicting transactions. Hence, we define two more levels of granularity for read-/write sets. First, the *Column Read/Write Sets* of a transaction T denote the columns read/written by T. Consider transaction W_2 from Table I. W_2 reads the column $S.B5$ (its column read set) and only updates $S.B2$ (its column write set). Similarly, the column write set of W_5 is $\{S.B4\}$, which is disjoint from the column write set of W_2. Hence, even though W_2 and W_5 modify the same table, they modify different columns and could be executed in parallel without affecting consistency.

Finally, the *Row Read/Write Sets* of a transaction T denote the rows read/written by T based on a primary key or a unique key. For instance, transaction W_1 (see Table I) updates the row in table R for which $A_1 = 100$ (A_1 is the primary key of R), whereas W_4 deletes the row for which $A_1 = 120$. Since W_1 and W_4 operate on different rows of the same table, they can also run concurrently without affecting consistency. We restrict the row sets to include only primary or unique key equality predicates as those are most popular, simple to identify, and efficient to compare against each other.

Based on the scope by which an SQL statement affects a table R, we categorize it in one of three *affecting classes*:

- **Row Affecting Statement (RAS)** when it modifies or accesses particular rows in R;
- **Column Affecting Statement (CAS)** when it modifies or accesses some columns of R;
- **Table Affecting Statement (TAS)** when it modifies or accesses all columns of R.

Tables I and II showcase several SQL statements along with their corresponding read/write sets and affecting classes.

B. Statement Replication Procedure

For each transaction T, a *Transaction State* (or *TState*) is built and maintained at the Transactions Buffer. A TState contains: (i) a TState identifier (*TSID*) that uniquely identifies T and is determined by the transaction commit timestamps; (ii) the *SQL write statements* of T in the order of execution; (iii) the read/write (R/W) sets of each statement and the overall T; and (iv) the completion operation: `commit` or `rollback`;

Hihooi's Extractor service receives the completed TStates from the Transactions Buffer and executes them on the local Extension DB. Its goal is to execute in parallel as many transactions as possible while ensuring that the local database replica is consistent with the Primary DB. Our approach utilizes the R/W sets of write transactions to determine whether some transactions affect the same data items in the database. If they don't, we say they are *independent*. In particular, when the corresponding table or column or row R/W sets of two write transactions are disjoint, they are independent.

One important property of the R/W sets is their *cumulative* nature. That is, if we take the union of the R/W sets of multiple statements, we get the R/W sets of a multi-statement transaction with the same correct semantics. Similarly, we can

Algorithm 1 Parallel execution of transactions at Extension DBs

```
runningState            ▷ combined state of running transactions
waitingState            ▷ combined state of waiting transactions
waitQueue               ▷ FIFO queue with waiting transaction states
function ONNEWTRANSACTION(tsNew)
    if areIndependent(runningState, tsNew) &
       areIndependent(waitingState, tsNew) then
        runningState.merge(tsNew)
        execute(tsNew)
    else
        waitingState.merge(tsNew)
        waitQueue.enqueue(tsNew)
    end if
end function
function ONTRANSACTIONCOMPLETE(tsOld)
    runningState.remove(tsOld)
    while waitQueue.isNotEmpty() &
       areIndependent(runningState, waitQueue.peek()) do
        tsRun ← waitQueue.dequeue()
        waitingState.remove(tsRun)
        runningState.merge(tsRun)
        execute(tsRun)
    end while
end function
```

combine the R/W sets of two or more multi-statement transactions that are running in parallel to build a transaction state that represents all running statements. This combined state allows us to avoid checking whether a new transaction is independent with each currently running transaction. Instead, we only need to check whether the new transaction is independent with the combined running transaction state.

Algorithm 1 shows the two functions that constitute the parallel execution algorithm employed by the Extractor, explained using an example. Suppose the 5 transaction from Table I must be executed at an Extension DB in that order. Transactions W_1 and W_2 are independent and will execute in parallel, while W_3 is placed in the wait queue since it conflicts with W_1. Even though W_4 is independent from the two running transactions (W_1 and W_2), it is not independent from the waiting W_3 and, hence, will also be placed in the wait queue. W_5 can also run in parallel as it modifies a different table than W_1 and a different column than W_2. When W_1 completes, W_3 can execute as it is independent from the running W_2, followed by W_4. Finally, the Extractor notifies the Transaction Manager with the latest applied TSID in sequential order without gaps. Hence, the Transaction Manager is aware of up to which transaction has been replayed on the Extension DBs in sequential order.

IV. Transaction Routing and Load Balancing

The write transactions are executed on the Primary DB, are given a sequential TSID upon completion, and are replicated to the Extension DBs. An Extension DB is considered *consistent* if it has replicated all transactions executed on the Primary DB. Read transactions can safely be routed either to the Primary DB or to any consistent Extension DB for execution. However, the asynchronous replication of write transactions to the Extension DBs can result in a lag between the Primary DB and the Extension DBs. In such a scenario, read transactions must either wait for at least one Extension DB to become consistent (which introduces latency delays) or be redirected to the Primary DB (which increases the load on the Primary DB). In either case, performance and scalability can suffer.

Hihooi implements a novel transaction-level routing and load balancing algorithm that utilizes R/W sets for directing transactions to Extension DBs, even if they are not consistent with the Primary DB. The key idea is that it is safe to route a read transaction T to an Extension DB if the tables (or columns/rows) accessed by T will not be modified by the write transactions that have yet to execute on the Extension DB. Our approach uses three hash indexes for separately mapping tables, columns, and rows to the latest write transaction that modified them. Suppose the 5 write transactions of our running example shown in Table I completed their execution on the Primary DB. The table index will show that table S was last modified by transaction with TSID=15, while the column index will show that $S.B2$ was last modified by TSID=12.

The next step in the transaction-level load balancing is to determine which Extension DBs are consistent for running an incoming read transaction. The indexes can be used for finding the TSID of the last transaction that modified any of the data items accessed by an incoming read transaction. Consider transaction R_3 that accesses columns $B2$, $B3$, and $B5$ of table S. Based on the content of the hash indexes, only the relevant column $S.B2$ has been modified by transaction with TSID=12. Hence, R_3 can execute on any Extension DB that has applied transactions with TSID=12 or higher.

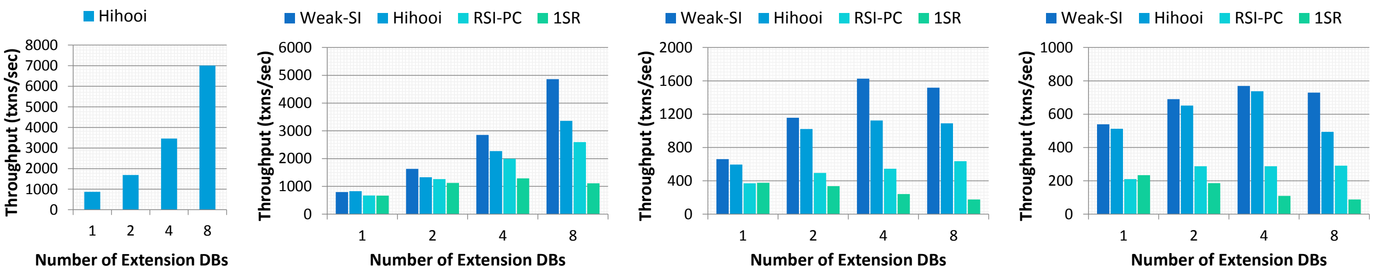
A. Consistency Levels

Most database engines (e.g., PostgreSQL, Oracle, DB2) use *snapshot isolation* (*SI*) for enforcing consistency [2]. With SI, each transaction operates on its own copy of data (a snapshot), allowing read transactions to complete without blocking. Similarly, database replication research has been focusing on SI and its variants, such as generalized SI, strong SI, and weak SI [16]. Hihooi works over a set of SI-based database replicas and offers the illusion of a single SI database to the client. Hence, it provides a form of **Global Strong Snapshot Isolation (GSSI)** [16].

By controlling the replication and routing mechanisms, Hihooi can offer three additional consistency levels at the granularity of a database session: (i) **Weak SI**: write transactions are asynchronously executed on the Extension DBs and read transactions are sent to any Extension DB regardless of their consistency; (ii) **Replicated SI with Primary Copy (RSI-PC)** provided by Ganymed [3]: write transactions are asynchronously executed on the Extension DBs and read transactions are sent to any Extension DB that is fully consistent with the Primary DB (but waits if none is available); (iii) **One-copy Serializability (1SR)** provided by C-JDBC [7]: write transactions are synchronously executed on all Extension DBs and read transactions are sent to any Extension DB.

V. Experimental Evaluation

The purpose of our evaluation is to evaluate the system's performance, scalability, and elasticity under varying workload



(a) Read-Only (b) Read-Heavy (c) Balanced (d) Write-Heavy

Fig. 2: OLTP workload scalability for TPC-C for different workload mixes and consistency levels

types and consistency levels. All experiments were run on a 13-node cluster running CentOS Linux 7.2 with 1 Primary DB, 1 Backup DB, 8 Extension DBs, and 3 client nodes (with 8-core, 3.2GHz CPU, 64GB RAM, and 2.1TB of HDD storage per node). For our evaluation, we used the industry standard OLTP workload **TPC-C** [17] with up to 48 clients, which contains complex and write-intensive transactions. The TPC-C database was populated with 500 warehouses for a total size of 50GB. The databases were fully replicated to the Extension DBs. We used PostgreSQL version 9.5.3 in all nodes.

A. OLTP Workload Scalability

This section studies the effectiveness and efficiency of Hihooi in scaling OLTP workloads by measuring their throughput and latency as we increase the number of Extension DBs. The comparison is done along two dimensions: (i) for different read/write workload mixes (i.e., Read-Only, Read-Heavy, Balanced, and Write-Heavy) and (ii) for different consistency levels (i.e., Weak-SI, Hihooi, RSI-PC, and 1SR; recall Section IV-A). Weak-SI is used to show the upper limit of performance that any system with consistency guarantees could achieve. RSI-PC is used by Ganymed, a similar middleware system that does not offer the type of replication and routing algorithms that Hihooi boasts, while 1SR (used by C-JDBC) shows the effect of synchronous replication.

For TPC-C, the Read-Only, Read-Heavy, Balanced, and Write-Heavy workload mixes were set up as 100%, 95%, 85%, and 70% of read transactions, respectively. The Write-Heavy workload represents the default transaction mix of TPC-C. Figure 2 shows the throughput rates for our workload mixes and consistency levels for TPC-C. The Read-Only workload scales linearly as the number of replicas increases; that is, the throughput doubles each time the number of Extension DBs doubles. As no writes are performed, there is no difference between the 4 consistency levels. The trend is similar for the Read-Heavy workload, with the exception of 1SR after 4 or more replicas are used. This is expected since the system has to wait for more replicas to apply all modifications before being able to serve any subsequent reads. As the percentage of writes increases in the workload, scalability naturally suffers for all consistency levels, since all writes are executed on the Primary DB and more reads have to wait for a consistent replica. Nonetheless, Hihooi is always able to offer comparable performance to Weak-SI and up to 2.6x and 6.7x higher throughput compared to RSI-PC and 1SR, respectively.

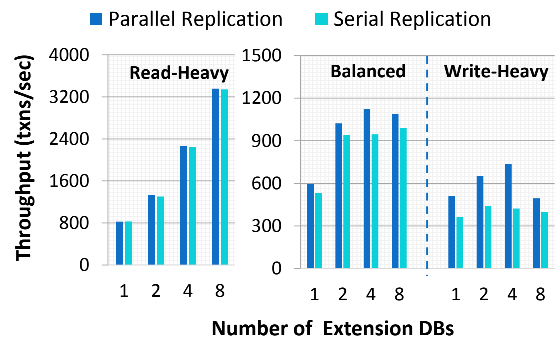


Fig. 3: Effect of parallel replication algorithm on TPC-C

B. Parallel Replication Algorithm

This section studies the performance implications of the parallel replication algorithm described in Section III-B compared to the common approach that executes the write transactions serially on the replicas. The two approaches have no to little impact to the throughput of the Read-Only and Read-Heavy TPC-C workloads (see Figure 3) since very few writes are applied to the replicas. Note that TPC-C contains 1 TAS and 11 RAS write statements, which are amenable to parallelism. As the percentage of writes increases for the Balanced and Write-Heavy workloads, the parallel algorithm has a profound effect on the throughput (up to 1.7x higher compared to the serial version) because it enables the Extension DBs to reach consistency quicker and, hence, be available to serve more reads. There is a drop in performance for the write-heavy workload on 8 Extension DBs because the writes in the scaled up workload overload the Primary DB.

C. Adding and Removing Extension DBs

Finally, we explore the scenario of adding and removing an Extension DB at run time. We started running the Balanced workload on Hihooi with 3 Extension DBs. After 20 minutes, we removed 1 Extension DB to simulate a failure or planned maintenance operation. Hihooi continued serving the workload without any issues due to its fault tolerant features, but with a 15% lower throughput, as shown in Figure 4. After 10 minutes, we restored the Extension DB and observed the throughput

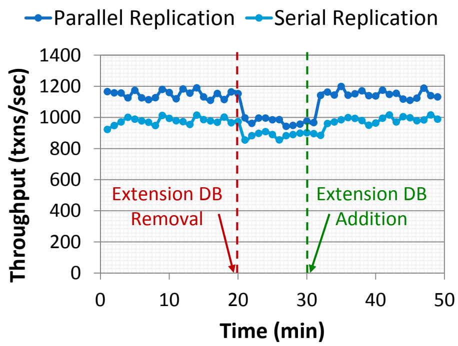


Fig. 4: Balanced TPC-C workload throughput after removing and adding an Extension DB

rate return to its normal level quickly. The Extension DB was able to serve its first read just 68 seconds after restoration due to our fine-grained routing algorithm, while it was able to apply all changes it missed during the outage in 85 seconds. We repeated the above procedure using the serial replication approach and observed a lower throughput during the entire experiment, while it took the Extension DB 125 seconds to catch up; highlighting once again the benefits of our parallel replication algorithm.

VI. Related Work

Database replication comes in two forms: master-slave and multi-master. Each form can be implemented either inside the database kernel or outside in a middleware layer. The former approach is heavily invasive and database-engine specific [18]. The middleware approach, also employed by Hihooi, leads to a seamless separation of concerns, supports unmodified database systems, and can enable heterogeneous environments. The Postgres-R [6] multi-master replication system enabled scalability and 1-copy-serializability, while a later version offered snapshot isolation [19]. Middle-R [4] was the middleware extension of Postgres-R but still required heavy database modifications. C-JDBC [7] is also a multi-master middleware that offers consistency guarantees through table-level locking at the middleware level. One of the manners that Hihooi differs from the state of the art is its new architecture that uses an in-memory distributed storage system for statement replication, rather than relying on command logging propagation or complex group communication protocols [2], [16].

Ganymed [3] is a similar master-slave middleware system that instead blocks a read transaction at the middleware layer until at least one replica becomes consistent. On the contrary, Hihooi never blocks any read transactions. Rather, it uses the transaction R/W sets to find the replicas that are consistent for each read transaction to run on. Similar to Ganymed, Pgpool-II [20] is another PostgreSQL-specific replication middleware solution that ships and applies WAL entries to the replicas. KuaFu [21] is a primary-backup, row-based replication system that also offers concurrent log replay by constructing and utilizing a graph to track write-write dependencies in the log; unlike Hihooi that relies solely on TSIDs and R/W sets. To allow read operations to be served on backups, KuaFu introduces barriers every N transactions to create snapshots that are consistent with some past states on the primary, unlike Hihooi that never uses barriers.

VII. Conclusions

Hihooi's ability to provide workload scalability and elasticity to existing databases without sacrificing consistency is an important step towards creating auto-scaling database features. The parallel replication algorithm allows the replicas to reach consistency quicker both while the system is running and when new replicas are added into the system. The routing algorithm avoids any delays by load balancing read transactions to consistent replicas and can easily adapt to a changing number of replicas. Given the efficient mechanisms already in place, we believe Hihooi can jump start interesting research directions towards workload-driven automated elasticity.

References

[1] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, "The Rise of Big Data on Cloud Computing: Review and Open Research Issues," *Information Systems*, vol. 47, pp. 98–115, 2015.
[2] E. Cecchet, G. Candea, and A. Ailamaki, "Middleware-based Database Replication: The Gaps between Theory and Practice," in *Proc. of the ACM SIGMOD*. ACM, 2008, pp. 739–752.
[3] C. Plattner and G. Alonso, "Ganymed: Scalable Replication for Transactional Web Applications," in *Proc. of MIDDLEWARE*. Springer-Verlag New York, Inc., 2004, pp. 155–174.
[4] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "MIDDLE-R: Consistent Database Replication at the Middleware Level," *ACM Trans. Comput. Syst.*, vol. 23, no. 4, pp. 375–423, 2005.
[5] M. Ronstrom and L. Thalmann, "MySQL Cluster Architecture Overview," MySQL, Tech. Rep., 2014.
[6] B. Kemme and G. Alonso, "Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication," in *Proc. of the 26th VLDB*. Morgan Kaufmann Publishers Inc., 2000, pp. 134–143.
[7] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible Database Clustering Middleware," in *Proc. of USENIX ATC*. USENIX, 2004, pp. 9–18.
[8] S. Elnikety, S. Dropsho, and F. Pedone, "Tashkent: Uniting Durability with Transaction Ordering for High-performance Scalable Database Replication," *ACM SIGOPS Review*, vol. 40, no. 4, pp. 117–130, 2006.
[9] J. Gray, P. Helland *et al.*, "The Dangers of Replication and a Solution," *ACM SIGMOD Record*, vol. 25, no. 2, pp. 173–182, 1996.
[10] R. Cattell, "Scalable SQL and NoSQL Data Stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.
[11] A. Thomson *et al.*, "Calvin: Fast Distributed Transactions for Partitioned Database Systems," in *Proc. of ACM SIGMOD*. ACM, 2012, pp. 1–12.
[12] K. Grolinger *et al.*, "Data Management in Cloud Environments: NoSQL and NewSQL Data Stores," *JoCCASA*, vol. 2, no. 1, p. 22, 2013.
[13] MemSQL, "MemSQL: The Database for Real-time Applications," 2018. [Online]. Available: https://www.memsql.com/
[14] M. Stonebraker and A. Weisberg, "The VoltDB Main Memory DBMS," *IEEE Data Eng. Bull*, vol. 36, no. 2, pp. 21–27, 2013.
[15] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 2004, no. 124, p. 5, 2004.
[16] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware Based Data Replication Providing Snapshot Isolation," in *Proc. of the ACM SIGMOD*. ACM, 2005, pp. 419–430.
[17] "TPC-C Benchmark, Revision 5.11.0," 2010, http://www.tpc.org/tpcc/.
[18] B. Kemme and G. Alonso, "Database Replication: A Tale of Research Across Communities," *PVLDB*, vol. 3, no. 1-2, pp. 5–12, 2010.
[19] S. Wu and B. Kemme, "Postgres-R (SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation," in *Proc. of the 21st ICDE*. IEEE, 2005, pp. 422–433.
[20] J. Maymala, *PostgreSQL for Data Architects*. Packt Publishing, 2015.
[21] M. Yang *et al.*, "KuaFu: Closing the Parallelism Gap in Database Replication," in *Proc. of the 29th ICDE*. IEEE, 2013, pp. 1186–1195.