Attaining Workload Scalability and Strong Consistency for Replicated Databases with Hihooi

Michael A. Georgiou
mica.georgiou@edu.cut.ac.cy
Cyprus University of Technology
Limassol, Cyprus

Michael Panayiotou
ms.panayiotou@edu.cut.ac.cy
Cyprus University of Technology
Limassol, Cyprus

Lambros Odysseos
lambros.odysseos@cut.ac.cy
Cyprus University of Technology
Limassol, Cyprus

Aristodemos Paphitis
am.paphitis@edu.cut.ac.cy
Cyprus University of Technology
Limassol, Cyprus

Michael Sirivianos
michael.sirivianos@cut.ac.cy
Cyprus University of Technology
Limassol, Cyprus

Herodotos Herodotou
herodotos.herodotou@cut.ac.cy
Cyprus University of Technology
Limassol, Cyprus

ABSTRACT

Database replication can be employed for scaling transactional workloads while maintaining strong consistency semantics. However, past approaches suffer from various issues such as limited scalability, performance versus consistency tradeoffs, and requirements for database or application modifications. Hihooi is a new replication-based master-slave middleware system that is able to overcome the aforementioned limitations. The novelty of Hihooi lies in its modern architecture as well as its replication and transaction routing algorithms. In particular, Hihooi replicates all write statements asynchronously and applies them in parallel at the replica nodes, while ensuring replica consistency. At the same time, a fine-grained transaction routing algorithm ensures that all read transactions are load balanced to the replicas consistently. This demonstration will showcase the key functionalities of Hihooi, including (i) practical management of system components and databases (e.g., add a new replica node), (ii) increased scalability compared to state-of-the-art approaches, and (iii) support for elasticity by suspending and resuming database replicas online without service interruption.

CCS CONCEPTS

• **Information systems → Middleware for databases**; **Data replication tools**; **Database transaction processing**.

KEYWORDS

database replication; transaction processing; workload scalability; concurrency control

ACM Reference Format:
Michael A. Georgiou, Michael Panayiotou, Lambros Odysseos, Aristodemos Paphitis, Michael Sirivianos, and Herodotos Herodotou. 2021. Attaining Workload Scalability and Strong Consistency for Replicated Databases with Hihooi. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3448016.3452746

SIGMOD '21, June 18–27, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3452746

1 INTRODUCTION

The sustained growth of internet-based services and applications are driving the rapid increase and high variability of workloads experienced by the underlying database systems. However, popular relational database systems (e.g., PostgreSQL, MySQL) have little to no ability for handling increasing workload demands gracefully (i.e., cannot offer *workload scalability*) or handling workload variations automatically (i.e., cannot offer *workload-driven elasticity*) [7].

Database replication has been employed in the past for increasing performance and availability of databases by fully replicating data across multiple nodes [1]. There are two replication variants, namely *multi-master* and *master-slave*. In the former, the write and read transactions are executed on all replicas, which rely on group communication primitives to agree on a serializable execution order of transactions, limiting the system's scalability [2, 4, 8]. In the latter, all write transactions are executed on one primary replica while the read transactions are processed by the other replicas [12, 13]. As long as the master node can handle the write workload, the system can scale linearly with the addition of more slave nodes [1].

Current database replication approaches suffer from various issues. MySQL Cluster [14] uses a synchronous replication mechanism that limits scalability. Postgres-R [8], and its middleware extension Middle-R [12], require engine modifications of PostgreSQL for extracting and applying tuple-based updates to replicas. Pgpool-II [11] and Ganymed [13] are master-slave replication middleware that apply all modifications serially at the replicas, which often causes replicas to fall behind the primary. Moreover, Ganymed blocks reads until a replica becomes consistent with the primary.

Hihooi is a new replication-based master-slave middleware system that is able to overcome the aforementioned limitations and offer workload scalability, strong consistency, and elasticity for transactional databases [6, 7]. Any database can easily become the master in a Hihooi deployment. Replication is then used to (i) increase the processing capacity of the system, thereby increasing throughput, and (ii) spread the load across the nodes, thereby decreasing latency. As a middleware system, Hihooi sits between the database engines and the application, and does not require any modifications for either one as it uses the industry's standard of JDBC/ODBC drivers.

Hence, Hihooi masks the complexity of the underlying replication process and offers a single database view in a practical manner.

The novelty of Hihooi lies in its modern architecture as well as its replication and transaction routing algorithms. In particular, Hihooi replicates all write statements *asynchronously* and applies them in *parallel* at the replica nodes, while ensuring replica consistency. At the same time, a *fine-grained transaction routing* algorithm ensures that all read transactions are load balanced to the replicas consistently. When Hihooi manages a set of Snapshot Isolation (SI)-based database replicas, it provides *Global Strong Snapshot Isolation*, i.e., it offers the illusion of a single SI database to the client. Finally, elasticity is achieved by supporting an easy and quick way to add and remove replicas from the cluster.

Contributions and Demo: We demonstrate Hihooi, a replication-based middleware system running on a cluster of nine PostgreSQL database nodes. Our demonstration of Hihooi aims at (i) introducing the Hihooi architecture and algorithms through a middleware system implementation and (ii) demonstrating the benefits of using Hihooi on top of existing transactional databases, in terms of both increased scalability and better elasticity support. We visually demonstrate the behavior of its core components in a range of scenarios, giving the audience members a complete visual insight into the behavior of Hihooi. In particular, the audience will have the ability to interact with the system through a graphical web interface for:

- managing the system components and databases, including adding a new replica node or a new database, and extending a database replication group;
- monitoring and comparing the workload scalability against two other state-of-the-art approaches, while variable transactional workloads are executing; and
- examining Hihooi's elasticity by suspending and resuming database replicas online and without service interruption, while observing the performance impact on transactional workloads.

2 DB REPLICATION WITH HIHOOI

Hihooi [6, 7] facilitates scalable and efficient transaction processing via replicating databases across multiple compute nodes and implementing novel replication and transaction routing algorithms, outlined below.

2.1 System Architecture

The Hihooi architecture is shown in Figure 1, along with the core components and the flow of transactions through the system. As a middleware system, Hihooi provides database-independent connectivity between the applications and the underlying database engines through the use of custom **Hihooi JDBC/ODBC Drivers** that implement the **Hihooi API**. Internally, Hihooi also uses JDBC drivers for interacting with the databases in order to execute the queries and to manage replication behind the scenes. Hence, neither the applications nor the database engines require any modifications to work with Hihooi. **HConsole** is an interactive console application that can be used for configuring and managing Hihooi, including adding/removing replicas and creating/editing database replication

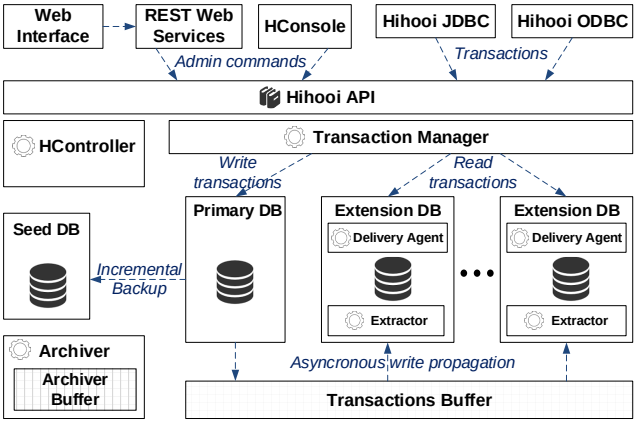

Figure 1: Hihooi middleware system architecture.

groups. The same functionality is exposed via **REST Web Services** and visualized by the Hihooi **Web Interface**.

The **Transaction Manager** (TM) is responsible for intercepting and handling all transactions. The write transactions are executed on the **Primary DB**, while the read transactions are load balanced to consistent **Extension DBs** (see Section 2.3). In addition, the TM manages the client sessions, oversees the available Extension DBs, and keeps track of which write transactions they have applied. Once a write transaction completes on the Primary DB, the transaction's statements are pushed into the **Transactions Buffer**, which is distributedly stored in memory using Memcached [5]. The Transactions Buffer acts as a highly available and fault tolerant propagation medium for all database modifications, which need to be applied asynchronously to the Extension DBs.

Each Extension DB node hosts one Extractor and one Delivery Agent service. The **Extractor** receives the new write transactions from the Transactions Buffer and applies them to the local database. The Extractors implement a novel algorithm for executing the transactions in parallel, while respecting the order imposed by the transaction commit timestamps on the Primary DB (described in Section 2.2). The **Delivery Agent** is responsible for executing the read-only transactions 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 TM.

The **Archiver** is responsible for performing incremental backups of the Primary DB to create the **Seed DB**, while keeping track with which transactions are included in the backup. Hence, the Seed DB represents a consistent checkpoint of the Primary DB at some point in time, and it is used for initializing new Extension DBs (see Section 2.4). In addition, the Archiver periodically moves the write transactions that have been applied by all Extension DBs from the Transactions Buffer to its local and persistent **Archiver Buffer** to keep the memory usage of the Transactions Buffer bounded. Finally, all system management operations, such as adding and removing Extension DBs, are coordinated by the **HController**.

2.2 Replication Management

Each Extension DB replica must reflect a transaction-consistent snapshot of the data at the Primary DB, in order to ensure that the

read transactions executing at the replica see a consistent view of the database. To retain global system consistency, Hihooi captures the total order of transaction completions on the Primary DB and replicates the write statements to the Extension DBs, while ensuring that each replica applies them in the same order. The statement replication takes place *asynchronously* to avoid delaying the write transactions executing at the Primary DB.

The conventional practice in database replication is to apply the writes serially at the slaves, even though the master processes them in parallel [10, 13]. With increasing writes, however, the lag between the master and a slave node can become significant [1]. Hihooi implements a novel algorithm (detailed in [7]) for applying write transactions in parallel at the slaves, while maintaining strong consistency guarantees. The algorithm utilizes Hihooi's notion of *transaction read/write sets*. Each transaction T will read and/or modify some tables in a database instance, defined as the *Table Read Set* and the *Table Write Set* of T, respectively. Similarly, the *Column* and *Row Read/Write Sets* of a transaction T denote the columns and rows (based on primary key equality) read/written by T, respectively. Note that the read/write sets of T are built dynamically and incrementally at the Transaction Manager upon the submission of T. The read/write sets of two transactions can then be used to determine whether the transactions affect the same data items in the database, which in turn can be used to decide when to parallelize their execution. Intuitively, if two write transactions modify two different tables (or different columns/rows of the same table), we can safely execute them in parallel and let them commit in reverse order, without violating any consistency guarantees [7].

2.3 Transaction Routing

As a middleware system, Hihooi intercepts all incoming transactions and is tasked with routing them to the underlying database engines for execution. Both single- and multi-statement transactions are supported. Write transactions are always routed to the Primary DB, whereas read transactions can be safely routed either to the Primary DB or to any consistent Extension DB for execution. However, the asynchronous replication of write transactions can result in a lag between the Primary DB and the Extension DBs, which is worsened under heavy write loads. In such a scenario, read transactions must either block until one Extension DB becomes consistent (which introduces latency delays) or be redirected to the Primary DB (which further increases its load). In either case, performance and scalability can suffer.

Unlike other systems, Hihooi implements a novel routing algorithm that utilizes read/write sets for directing read 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. To achieve this, Hihooi keeps track of the completed transactions that have been applied on each of the Extension DBs along with the transactions that are currently running on the Primary DB. Hence, Hihooi recognizes which tables, columns, or rows are up-to-date on each of the Extension DBs. Next, Hihooi checks which read queries are safe (from a consistency point of view) to execute on which Extension DBs. In the case where multiple Extension DBs can execute an incoming query, Hihooi will perform load balancing and send the query to the least-loaded Extension DB. Hihooi is the first middleware system able to also do this for read queries that are part of multi-statement write transactions [7]. By default, Hihooi supports Global Strong Snapshot Isolation (GSSI) [9]. By controlling the replication and routing mechanisms, Hihooi can also offer Weak SI, Replicated SI with Primary Copy [13], and One-copy Serializability [2].

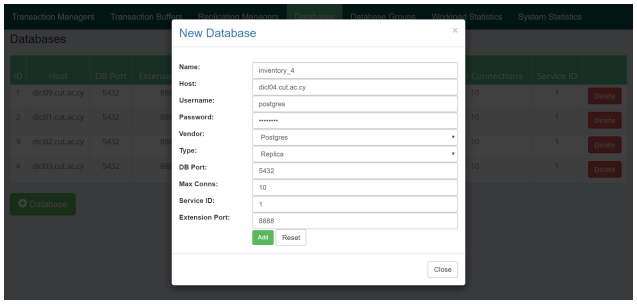

Figure 2: The *Databases* view.

2.4 Scalability Management

Hihooi maintains an incremental backup of the Primary DB, called Seed DB, in order to (i) enable fast recovery from Primary DB failures and (ii) add new Extension DBs efficiently without affecting the performance of the Primary DB or the existing Extension DBs. A new Extension DB is created by cloning the Seed DB into a new node, followed by the execution of all write transactions missed since the creation of the backup (located on the Archiver and Transactions Buffer). Since Hihooi already allows for Extension DBs to fall behind and uses a smart query routing algorithm for executing queries correctly, it is not necessary to enact a global barrier to ensure consistency [1]. Instead, as soon as the Extension DB is created, it can join the system and start executing read transactions, while concurrently applying the write transactions. When an Extension DB is removed, the Transaction Manager automatically re-routes its read transactions to other consistent Extension DBs.

3 DEMONSTRATION PLAN

The demonstration will showcase the key functionalities and benefits offered by Hihooi, which include (i) practical system and database management (e.g., add a new Extension DB node or a new database replica), (ii) better workload scalability compared to other state-of-the-art approaches, and (iii) support for elasticity, i.e., suspending and resuming database replicas online. For the purposes of the demonstration, Hihooi will be running on our 9-node in-house cluster. Each node has an 8-core, 2.4GHz CPU, 24GB RAM, 1.5TB HDD drives, and runs PostgreSQL v9.5.3. Workloads from the popular TPC-C [15] and YCSB [3] benchmarks will be executed on the cluster so that the audience can experience the behavior of Hihooi and get a better understanding of its advantages. At the same time, a poster will be used to introduce the audience to the Hihooi architecture and algorithms.

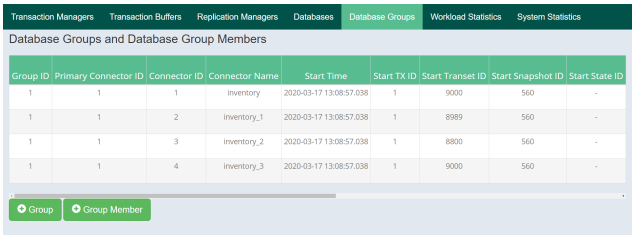

Figure 3: The *Database Groups* view.

3.1 Practical System and Database Management

For the first demonstration part, the audience will be able to interact with the various management features of the Hihooi Web Interface and experience first hand the ease with which the components can be configured. The Web Interface contains three views for managing core components of the middleware system as well as two views for managing databases and database replication groups. In particular, the *Transaction Managers*, *Transactions Buffers*, and *Replication Managers* views can be used for adding and configuring the corresponding Hihooi component with the same name. Note that a Replication Manager is a service that combines the Extractor and Delivery Agent components discussed in Section 2.1. While a single Hihooi deployment has one Transaction Manager, one Transactions Buffer, and several Replication Managers, the Web Interface has been built to manage multiple Hihooi deployments.

One Hihooi deployment can be used for managing and replicating multiple databases, organized into groups. A *database replication group* associates a primary database with a set of replica databases. The *Databases* view lists information about the current databases (both primary and replica databases), with the options of adding a new database or deleting an existing one. Figure 2 shows a screenshot of the *Databases* view when adding a new database. The user needs to fill in the necessary information and select whether this database will serve as a primary one (and hence can be replicated) or as a replica of an existing primary database. Creating a database group and associating replicas with a primary database can be handled in the *Database Groups* view, shown in Figure 3.

3.2 Workload Scalability

In this part of the demonstration, we plan to execute a data-intensive workload, while varying the number of concurrent clients (and hence, the total workload), the ratio of read-to-write transactions, and the number of available replicas. In this way, the audience will get a better understanding of how quickly the write transactions are applied to the replicas and how the read transactions are load balanced across the replicas. At the same time, we will compare the scalability of Hihooi against the approaches employed by two other middleware systems, namely Ganymed [13] and C-JDBC [2], and showcase Hihooi's superior performance.

The workload statistics and system utilization across all nodes will be visualized using the *Workload Statistics* and *System Statistics* views, respectively. The provided workload information includes read and write transactions per second for the overall Hihooi deployment, as well as for the primary and replica databases. The information is presented in line graphs that are continuously updated

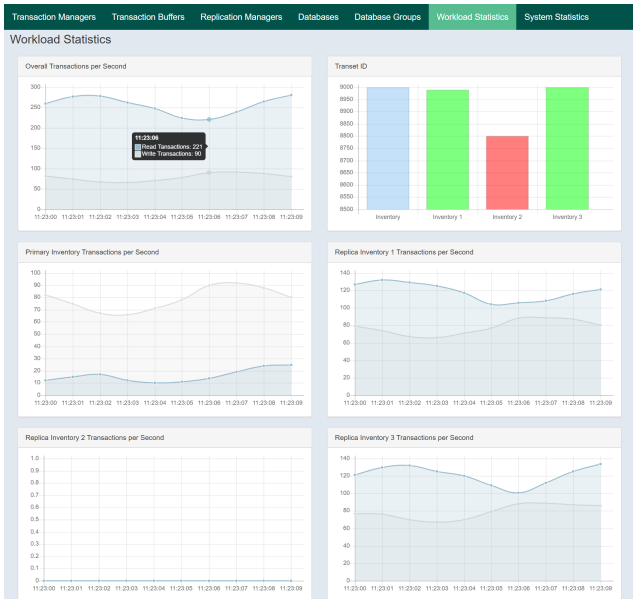

Figure 4: The *Workload Statistics* view.

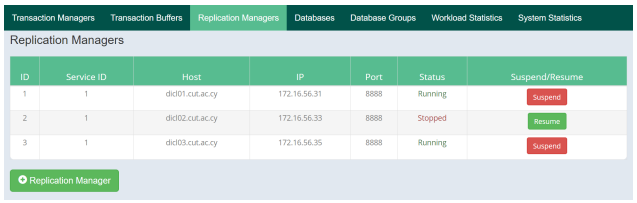

Figure 5: The *Replication Managers* view.

live, as illustrated in Figure 4. In addition, a bar graph (top-right part of Figure 4) shows the ID of the latest committed transaction on each database, allowing the user to visually see whether there is any significant lag between the primary and the replica databases. In this particular example, the second replica is currently offline (shown in red), while the other two replicas (shown in green) are almost up to date with the primary (shown in blue). The *System Statistics* view shows live metrics regarding CPU, memory, disk, and network utilization per node.

3.3 System Elasticity

Continuing with the workload executions from Section 3.2, the audience will also observe how the workload and system behaves when a Replication Manager (RM) is suspended, resumed, or added to the system online without any interruptions; thus getting a deeper understanding of the elastic capabilities of Hihooi. In particular, when an RM is suspended (or lost due to some failure), its workload is automatically re-routed to other consistent replicas. When the RM is resumed, the missed write transactions are replayed from that point forward, while the RM starts serving read transactions shortly after. The suspending, resuming, and adding RMs functionalities are enabled by the *Replication Managers* view, shown in Figure 5, which also visualizes information and the status of the RMs.

REFERENCES

[1] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. 2008. Middleware-based Database Replication: The Gaps between Theory and Practice. In *Proc. of 2008 Intl. Conf. on Management of Data (SIGMOD)*. ACM, 739–752.

[2] Emmanuel Cecchet, Marguerite Julie, and Willy Zwaenepoel. 2004. C-JDBC: Flexible Database Clustering Middleware. In *Proc. of USENIX Annual Technical Conference (ATC)*. USENIX, 9–18.

[3] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of 1st ACM Symposium on Cloud Computing (SoCC)*. ACM, 143–154.

[4] Sameh Elnikety, Steven Dropsho, and Fernando Pedone. 2006. Tashkent: Uniting Durability with Transaction Ordering for High-performance Scalable Database Replication. In *Proc. of 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 117–130.

[5] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux Journal* 2004, 124 (2004), 5.

[6] Michael A Georgiou, Aristodemos Paphitis, Michael Sirivianos, and Herodotos Herodotou. 2019. Towards Auto-Scaling Existing Transactional Databases with Strong Consistency. In *Proc. of the IEEE 35th Intl. Conf. on Data Engineering Workshops (ICDEW)*. IEEE, 107–112.

[7] Michael A Georgiou, Aristodemos Paphitis, Michael Sirivianos, and Herodotos Herodotou. 2020. Hihooi: A Database Replication Middleware for Scaling Transactional Databases Consistently. *IEEE TKDE* Early Access (2020), 17 pages.

[8] Bettina Kemme and Gustavo Alonso. 2000. Don't be Lazy, be Consistent: Postgres-R, a New Way to Implement Database Replication. In *Proc. of the 26th Intl. Conf. on Very Large Databases (VLDB)*. Citeseer, 134–143.

[9] Yi Lin, Bettina Kemme, Marta Patiño-Martínez, and Ricardo Jiménez-Peris. 2005. Middleware Based Data Replication Providing Snapshot Isolation. In *Proc. of 2005 Intl. Conf. on Management of Data (SIGMOD)*. ACM, 419–430.

[10] Ludovic Marcotte. 2005. Database Replication with Slony-I. *Linux Journal* 2005, 134 (2005), 1.

[11] Jayadevan Maymala. 2015. *PostgreSQL for Data Architects*. Packt Publishing.

[12] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. 2005. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst* 23, 4 (2005), 375–423.

[13] Christian Plattner and Gustavo Alonso. 2004. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of ACM/IFIP/USENIX Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*. Springer, 155–174.

[14] Mikael Ronstrom and Lars Thalmann. 2014. *MySQL Cluster Architecture Overview*. Technical Report. MySQL. https://confluence.oceanobservatories.org/download/attachments/16418744/mysql-cluster-technical-whitepaper.pdf.

[15] TPC-C 2010. TPC-C Benchmark, Revision 5.11.0. http://www.tpc.org/tpcc/.