



Cyprus  
University of  
Technology

FACULTY OF  
ENGINEERING AND  
TECHNOLOGY

**Doctoral Dissertation**

**ENABLING WORKLOAD SCALABILITY, STRONG  
CONSISTENCY AND ELASTICITY WITH  
TRANSACTIONAL DATABASE REPLICATION**

**Michael A. Georgiou**

**Limassol, May 2020**



**CYPRUS UNIVERSITY OF TECHNOLOGY**  
**FACULTY OF ENGINEERING AND TECHNOLOGY**  
**DEPARTMENT OF ELECTRICAL ENGINEERING, COMPUTER**  
**ENGINEERING AND INFORMATICS**

**Doctoral Dissertation**

**ENABLING WORKLOAD SCALABILITY, STRONG**  
**CONSISTENCY AND ELASTICITY WITH**  
**TRANSACTIONAL DATABASE REPLICATION**

**Michael A. Georgiou**

Limassol, May 2020

# Approval Form

Doctoral Dissertation

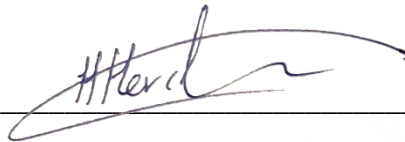
## **ENABLING WORKLOAD SCALABILITY, STRONG CONSISTENCY AND ELASTICITY WITH TRANSACTIONAL DATABASE REPLICATION**

Presented by

Michael A. Georgiou

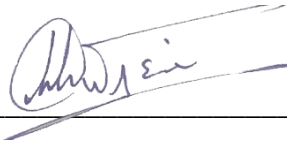
Supervisor: Herodotos Herodotou, Assistant Professor

Signature \_\_\_\_\_



Member of the committee: Andreas S. Andreou, Professor

Signature \_\_\_\_\_



Chair of the committee: Panos K. Chrysanthis, Professor

Signature \_\_\_\_\_



## **Copyrights**

Copyright © Year of dissertation submission Michael A. Georgiou

All rights reserved.

The approval of the dissertation by the Department of Electrical Engineering, Computer Engineering and Informatics does not imply necessarily the approval by the Department of the views of the writer.

## **Acknowledgements**

*This thesis is the culmination of my journey of Ph.D, a journey of exploration, research, and creativity. A journey of commitment and focus to your goal. During the project, your soul is constantly filled with lusts of faith to keep trying. On this journey, I was not alone, I had people who supported me and believed in me, and to these people I want to give my deepest thanks in depth of my heart.*

*Firstly, I would like to express my sincere gratitude to my advisor Dr. Herodotos Herodotou for the continuous support of my Ph.D study and related research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study.*

*I am greatly indebted to my research guide Dr. Michael Sirivianos. Under his guidance I successfully overcame many difficulties and learnt a lot.*

*With a special mention to Dr. Verena Kantere my initial research guide. This work would not have been possible without her initial guidance and involvement.*

*Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Panos Chrysanthis and Prof. Andreas Andreou, for their insightful comments and encouragement, but also for the hard question which incited me to widen my research from various perspectives.*

*I would always remember my colleague Aristodemos Paphitis for the fun-time we spent together, sleepless nights that gave us the courage to complete tasks before deadlines and for stimulating the discussions. His knowledge was very impressive.*

*Also, I would like to express my gratitude to my University who giving me the opportunity to conduct this research, with the concession of access to the laboratory and research facilities. Without it precious supports it would not be possible to conduct this research.*

*I am especially grateful to my parents. I always knew that you believed in me and wanted the best for me. Thank you for teaching me that my job in life was to learn, to be happy, and to know and understand myself; only then could I know and understand others.*

*I owe thanks to a very special person, my wife, Katerina for her continued and unflinching love, support and understanding during my pursuit of Ph.D degree that made the completion of thesis possible. You were always around at times I thought that it is impossible to continue, you helped me to keep things in perspective. I greatly value her contribution and deeply appreciate his belief in me. I appreciate my sons Andreas and Stavros for the patience that they showed during my thesis. Words would never say how grateful I am you. I consider myself the luckiest in the world to have such a lovely and caring family, standing beside me with their love and unconditional support.*

*I cannot forget to thanks, Mrs Janet Panteli , who helped me in proofreading the English language texts during my research.*

*Finally, a big thanks to my dog UMI, for her strong consistency and unconditional love who companion me in my home study.*

*Michael @ Georgiou*

*Dedicated*  
*To my parents Andreas and Koulla.*  
*To my wife Katerina and my sons Andreas and Stavros.*

## Abstract

The proliferation of internet-based services and applications has led to both a rapid growth and high variability of transactional workloads, which can negatively affect 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). Database replication has been successfully employed in the past to scale performance and improve availability of relational databases but current approaches suffer from various issues including limited scalability, performance versus consistency tradeoffs, and requirements for database or application modifications. This thesis presents a new replication-based middleware system, called Hihooi, which is able to achieve workload scalability, strong consistency, and elasticity for existing transactional databases at a low cost. 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, which is used to increase throughput (via increasing the processing capacity of the system) and decrease latency (via spreading the load across the nodes). The novelty of Hihooi lies in 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 that all replicas remain consistent with the primary copy. At the same time, a fine-grained transaction routing algorithm ensures that all read transactions are load balanced to the replicas, while maintaining strong consistency semantics. Finally, elasticity is achieved by supporting an easy and quick way to add and remove replicas from the cluster. A thorough experimental evaluation with several well-established benchmarks highlights how Hihooi is able to achieve almost linear workload scalability for existing transactional databases.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Research Contributions . . . . .	4
<b>2</b>	<b>Preliminaries on Concurrency Control, Scalability, and Elasticity</b>	<b>5</b>
2.1	Concurrency Control . . . . .	5
2.2	Scalability . . . . .	8
2.2.1	How RDBMSs Scale Horizontally . . . . .	15
2.3	Elasticity . . . . .	18
2.4	Discussion . . . . .	20
<b>3</b>	<b>Database Replication for Distributed Databases</b>	<b>22</b>
3.1	Concurrency Control for Replicated Databases . . . . .	22
3.1.1	Serializability . . . . .	23
3.2	Replication protocols . . . . .	25
3.2.1	Eager Replication Protocols . . . . .	27
3.2.2	Lazy Replication Protocols . . . . .	28
3.3	Group Communication . . . . .	29
3.4	Middleware Based Replication . . . . .	30
<b>4</b>	<b>Hihooi System Overview</b>	<b>32</b>
<b>5</b>	<b>Hihooi System Architecture</b>	<b>35</b>
<b>6</b>	<b>Database Replication with Hihooi</b>	<b>38</b>
6.1	Transaction Read/Write Sets . . . . .	38
6.2	Hihooi Commit/Rollback Phase . . . . .	42
6.3	Statement Replication Procedure . . . . .	43
6.4	Resolutions against Statement Based Replication Challenges . . . . .	51
6.5	Hihooi and Primary DB Consistency Challenges . . . . .	54

<b>7</b>	<b>Concurrency Control</b>	<b>56</b>
7.1	Transaction-level Load Balancing . . . . .	56
7.2	Statement-level Load Balancing . . . . .	59
7.3	Consistency Levels . . . . .	59
<b>8</b>	<b>Scalability Management</b>	<b>63</b>
8.1	Hihooi Startup and Planned Shutdown . . . . .	63
8.2	System Initialization, Backups and Fault Recovery . . . . .	64
8.3	Adding and Removing Extension DBs . . . . .	68
8.4	Towards Replica Self-Management . . . . .	69
<b>9</b>	<b>Experimental Evaluation</b>	<b>71</b>
9.1	OLTP Workload Scalability . . . . .	71
9.2	OLTP-OLAP Workload Scalability . . . . .	77
9.3	Effect of Affecting Classes . . . . .	78
9.4	Parallel Replication Algorithm . . . . .	79
9.5	Statement-level Load Balancing . . . . .	80
9.6	Adding and Removing Extension DBs . . . . .	81
9.7	Transactions Buffer Failure and Recovery . . . . .	81
9.8	Comparison with PostgreSQL Replication . . . . .	82
<b>10</b>	<b>Preliminary Investigation of Future Work</b>	<b>84</b>
10.1	The Hihooi Autonomic Elasticity Model . . . . .	84
10.2	Metrics Definitions . . . . .	88
10.3	Discussion . . . . .	90
<b>11</b>	<b>Comparison with State-of-the-Art</b>	<b>92</b>
<b>12</b>	<b>Conclusions</b>	<b>96</b>
<b>13</b>	<b>Appendix</b>	<b>97</b>

## List of Tables

1	Example write transactions on tables R(A1*, A2, A3, A4) and S(B1*, B2, B3, B4, B5) along with corresponding write sets, read sets, affecting classes, and transaction state identifiers (TSIDs) . . . . .	39
2	Definitions of read/write sets for relational algebra operations. The read sets of write operations equal the corresponding read sets of expression $E$	40
3	Example read transactions on tables R(A1*, A2, A3, A4) and S(B1*, B2, B3, B4, B5) along with the corresponding read sets, affecting classes, and consistent transaction state identifiers (TSIDs) . . . . .	40
4	Summary of Practical issues and resolutions for statement replication . . .	51
5	The content of the Tables, Columns, and Rows Hash Indexes after executing the Table 1 transactions . . . . .	58
6	Composition of TPC-C workload mixes . . . . .	72
7	Percentage (%) of TAS, CAS, and RAS statements . . . . .	78

## List of Figures

1	RDBMS Multi-Master Horizontal Scaling . . . . .	16
2	RDBMS Master-Slave Horizontal Scaling . . . . .	16
3	RDBMS Share-All Horizontal Scaling . . . . .	17
4	Replication Execution Model . . . . .	22
5	Eager Single Master Replication Protocol Actions (1) A Write is applied on the master copy; (2) Write is then propagated to the other replicas; (3) Updates become permanent at commit time; (4) A Read goes to any slave copy. Figure is adapted from [71] chapter 13. . . . .	28
6	Lazy Single Master Replication Protocol Actions (1) Update is applied on the local replica; (2) Transaction commit makes the updates permanent at the master; (3) Update is propagated to the other replicas in refresh transactions; (4) Transaction 2 reads from local copy. Figure is adapted from [71] chapter 13. . . . .	29
7	Hihooi Architecture . . . . .	36
8	Hihooi processing flow for executing single write and multi-write transactions . . . . .	42
9	Hihooi Commit/Rollback Flow for single write or multi-write transactions	43
10	Hihooi state snapshot at time $t_2$ . . . . .	67
11	OLTP workload scalability for TPC-C for different workload mixes and consistency levels . . . . .	72
	(a) Read-Only . . . . .	72
	(b) Read-Heavy . . . . .	72
	(c) Balanced . . . . .	72
	(d) Write-Heavy . . . . .	72
12	OLTP workload scalability for YCSB for different workload mixes and consistency levels . . . . .	73
	(a) Read-Only . . . . .	73
	(b) Read-Heavy . . . . .	73

	(c) Balanced . . . . .	73
	(d) Write-Heavy . . . . .	73
13	Average latency for TPC-C and YCSB for different workload mixes and consistency levels . . . . .	74
	(a) Read-Heavy . . . . .	74
	(b) Balanced . . . . .	74
	(c) Write-Heavy . . . . .	74
14	Mixed OLTP-OLAP workload scalability for CHB . . . . .	75
15	Effect of using the TAS, CAS, and RAS classes . . . . .	75
16	Effect of Hihooi's parallel replication algorithm on TPC-C . . . . .	76
17	Effect of statement-level load balancing on TPC-C . . . . .	76
18	YCSB throughput after removing & adding 1 Extension DB . . . . .	77
19	YCSB throughput during Transactions Buffer failure . . . . .	78
20	Pgpool-II Vs Hihooi for YCSB Balanced workload . . . . .	82
21	RAS Statements per second . . . . .	85
22	RAS Time per call . . . . .	85
23	Results after enabling Hihooi Autonomic Elasticity . . . . .	90

## Lists of Abbreviations

1SR.....	1 Copy Serialability
2PC.....	2 Phase Commit
A.C.I.D.....	Atomicity Consistency Isolation Durability
AMI.....	Amazon Machine Image
API.....	Application Program Interface
ARMA.....	Auto Regression Moving Average
BoT.....	Beginning of Transaction
C.A.P.....	Consistency Availability Partition Tolerance
CAS.....	Column Affected Class
CHB.....	The Mixed Workload CH Benchmark
DBaaS.....	Database as a Service
DICL.....	Data Intensive Computing Research Lab
DP.....	Data Processing
EC.....	Eventually Consistency
EoT.....	End of Transaction
EXT.....	Extension Host
EXT DB.....	Extension Database
FFT.....	Fast Fourier Transform
FIFO.....	First In First Out
GSSI.....	Global Strong Snapshot Isolation
ILP.....	Integer Linear Program
IaaS.....	Infrastructure as a Service
JDBC.....	Java Database Connectivity
MVCC.....	Multi-version Concurrency Control
ODBC.....	Open Database Connectivity
OLAP.....	Online Analytic Processing
OLTP.....	Online Transaction Processing
PRM.....	Primary Host

PRM DB.....	Primary Database
QoS.....	Quality of Service
RAC.....	Real Application Cluster
RAS.....	Row Affected Class
RDBMS.....	Relational Database Management System
RPS.....	Reads per second
RS.....	Read Set
SER.....	Replication with Serialability
SI.....	Snapshot Isolation
SLA.....	Service Level Agreement
SME.....	Small and Medium Enterprises
SQL.....	Structure Query Language
TAS.....	Table Affected Class
TE.....	Transaction Engine
TM.....	Transaction Manager
TPC.....	Time per call
TPC-C.....	Transaction Processing Performance Council Benchmark C
TPC-H.....	Transaction Processing Performance Council Benchmark H
TSID.....	Transaction Set Identifier
VM.....	Virtual Machine
VMC.....	Virtual Machine Controller
WAL.....	Write Ahead Log
WS.....	Write Set
YCSB.....	Yahoo Cloud Service Benchmark

# 1 Introduction

The recent explosion of data has led to the development of innovative systems for large-scale data processing [13]. In the transactional systems arena, systems such as Google’s Megastore [14] and Spanner [34] were introduced for handling massive amounts of data, even across datacenter boundaries. However, the majority of transactional databases are smaller than 1TB in size [38], indicating that excessive data scalability is a non-requirement for most small and medium enterprises (SMEs). Rather, modern applications tend to experience both rapid growth and variability of users (and consequently application workload) due to the advent of the Internet and Internet-connected devices. Therefore, *workload scalability*, i.e., the ability to handle increasing workload demands, as well as *support for elasticity* to handle variations in those workloads, are critical for *existing database instances*.

*NoSQL* technologies, such as MongoDB [29] and Cassandra [59], were explicitly designed to address scalability and elasticity requirements. By doing so, NoSQL systems sacrifice traditional consistency models and the familiarity of SQL. Hence, they cannot replace existing transactional database systems. More recently, a new class of systems has arisen, called *NewSQL*, that offers similar scalable performance as NoSQL while still maintaining the ACID guarantees of a traditional database system [47]. NewSQL systems, however, are often highly optimized for a narrow set of use cases (e.g., MemSQL[64] is tuned for clustered analytics) and require other compromises related to language support or transaction and workload handling capabilities (e.g., in VoltDB [96], the unit of transaction is a Java stored procedure). Finally, the evolution of cloud computing has led to several *Database-as-a-Service (DBaaS)* offerings (e.g., Amazon RDS, Azure SQL DB) that natively support scalability and elasticity in a pay-as-you-go model. Yet, SMEs are cautious in adopting them due to the high costs associated with rewriting applications and retraining employees, as well as privacy and security concerns [49].

A typical approach to scaling an existing database system is to *scale up*; i.e., to add more physical resources (e.g., memory, disks) to the server or upgrade to a higher-end server or a shared-disk database clustering solution (e.g., Oracle RAC [70]). Apart from



being very expensive due to both hardware and software licensing costs, such solutions necessitate over-provisioning for peak and eventual volumes [26]. The alternative is to utilize a *scale-out* approach, which can help reduce costs by hosting databases on multiple commodity hardware servers. *Data partitioning* (or *sharding*) is one of the two main scale-out physical implementations, based on which the database data is partitioned and spread across all nodes [24]. 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 [99].

The second scale-out approach is *database replication*, which has been used for increasing performance and availability of databases under various requirements [26]. This approach fully replicates data across all nodes and it comes in two forms, i.e., *multi-master* and *master-slave*. In the former, all replicas serve both read and write transactions but need explicit synchronization mechanisms in order to agree to a serializable execution order of transactions, so that each replica executes them in the same order [27, 55, 41]. Concurrent transactions might conflict, leading to aborts and thus limiting the system's scalability [46]. In *master-slave replication*, one primary copy handles all write operations while the other replicas process only read operations [77, 73]. As long as the master node can handle the write workload, the system can scale linearly with the addition of more slave nodes [26]. The biggest challenge here lies in the trade-off between performance and consistency of the overall system.

This manuscript 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 the processing capacity of the system (which increases throughput), to spread the load across the nodes (which decreases latency), and to mask potential failures of individual nodes (which improves availability). As a middleware system, Hihooi sits between the database engines and the client, 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. Load distribution, fault tolerance, and failure recovery are all handled by Hihooi.

The novelty of Hihooi lies in 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 that all replicas remain consistent with the primary copy. At the same time, a *fine-grained transaction routing* algorithm ensures that all read transactions are load balanced to the replicas, while maintaining strong consistency semantics. In particular, Hihooi supports *global strong snapshot isolation*, explained and proved in Section 7.3. Finally, elasticity is achieved by supporting an easy and quick way to add and remove replicas from the cluster (partly due to the master-slave architecture).

Existing replication-based approaches fall short of achieving all of Hihooi's aforementioned desiderata. Open-source solutions for replication are database-specific. MySQL Cluster [87] uses a synchronous replication mechanism which limits scalability. PostgreSQL [55] integrates replica control into the kernel of PostgreSQL and utilizes special multicast primitives to propagate low-level write operations to the replicas. Middle-R [73] allows all replicas to execute write transactions and uses a group communication system to determine a global commit order but requires database engine modifications for extracting and applying tuple-based modifications. Finally, Ganymed [77] is a master-slave replication middleware similar to Hihooi but applies all changes serially at the replicas and offers only a coarse-grained load balancing of transactions.

## 1.1 Research Contributions

In summary, the research contributions are:

1. Novel Transaction- and statement-level **routing algorithms** for executing read transactions consistently and efficiently.
2. A **statement replication algorithm** for applying writes in parallel while ensuring consistent replicas.
3. A new **database replication middleware architecture** for achieving workload scalability with strong consistency.
4. An **extensive evaluation** showcasing the workload scalability that is attainable with Hihooi.

Section 2 provides preliminaries on concurrency control, scalability and elasticity. Section 3 presents the important related to the Database Replication for Distributed Databases. Sections 4 and 5 provide an overview of Hihooi and its architecture, respectively. Section 6 presents Database Replication with Hihooi. Sections 7 and 8 describe Hihooi's Concurrency Control and Scalability Management, respectively. Section 9, presents the experimental evaluation. Section 10 provides a preliminary investigation for Autonomic Self-Management Database Elasticity. Section 11 presents the related work. Finally, Section 12 concludes the manuscript.

## **2 Preliminaries on Concurrency Control, Scalability, and Elasticity**

The majority of transactional databases are smaller than 1TB in size [38], indicating that excessive data scalability is a non-requirement for most small and medium enterprises (SMEs). Rather, modern applications tend to experience both rapid growth and variability of users (and consequently application workload) due to the advent of the Internet and Internet-connected devices. Therefore, workload scalability for enterprise workloads are critical for existing database instances. A workload can scale-up if the distributed database cluster has the ability to increase throughput as the number of cluster's nodes increases with the same factor as the workload size increases. This challenge requires (i) a distributed database cluster that should be easily and quickly able to provision and de-provision resources in an autonomic manner such that at each point in time the available resources match the current demand as closely as possible [50] and (ii) analysis of the workload characteristics in order to find out which distributed database cluster is best suited with this.

Enterprise workloads are (i) by nature transactional, and as a result, strong consistency is mandatory, (ii) are based on a relational model and as a result, SQL is mandatory, and (iii) deployed on customer's premises using a Traditional RDBMS. Based on these characteristics, in this section we examine the current database system technology taking into account the consistency, scalability and elasticity that various systems offer in order to found out which technology is best suited with our problem. Finally, we present each system's weaknesses and potentials.

### **2.1 Concurrency Control**

A Database transaction is a logical unit of work that takes the database from one consistent state to another. Transactions can terminate successfully(COMMIT) or unsuccessfully(ABORT). In the context of the Relational Database, transactions must satisfy the A.C.I.D properties which are intended to guarantee validity even in the event of errors,

power failures, etc. These properties are *Atomicity* (all of a transaction completes, or none of it does), *Consistency* (data is always valid according to schema constraints), *Isolation* (it describes the degree to which concurrent transactions are aware of each other e.g., by accessing the same data items), *Durability* (committed changes are not lost) [109]. Providing isolation is the main goal of *concurrency control*. Concurrency control is needed when multiple users are allowed to access the database simultaneously. Without it, problems of lost update, uncommitted dependency, and inconsistent analysis can arise.

The highest level of isolation between transactions is achieved via serializability. Transactions are said to be serializable if the results of running transactions simultaneously are the same as the results of running them serially, that is, one after the other. Generally, serializability is implemented using the two-phase locking mechanism. In two-phase locking, a transaction acquires all its locks before releasing any. Locks may be shared (read) or exclusive (write). Even if Serializability offers absolute correctness, it influences transactions' performance and typically leads to poor transaction execution rate and high transaction response time. As a result, Serializability is relaxed to weaker methods with many characteristics of full serializability, but still short of some, and unfit in many situations. For example, a *Multiversion Concurrency Control (MVCC)* is a common way to increase concurrency and performance by generating a new version of a database object each time the object is written, and allowing transactions' read operations of several last relevant versions (of each object); i.e., when a transaction attempts to read a data item, the system selects one of the versions that ensures serializability. Based on MVCC, *snapshot isolation (SI)* is another version of concurrency control that has gained importance in recent years and now provides as a standard consistency criterion in a number of commercial systems. SI allows read transactions (queries) to read stale data by allowing them to read a snapshot of the database that reflects the committed data at the time the read transaction starts. Consequently, the reads are never blocked by writes, even though they may read old data that may be dirtied by other transactions that were still running when the snapshot was taken. Hence, the resulting histories are not serializable, but this is accepted as a reasonable tradeoff between a lower level of isolation and better

performance. On the other hand, several proposals exist for modifying the application programs, without changing their semantics, so that they can achieve serializability even on an engine that uses SI with negligible reduction in throughput [9]. Oracle, DB2, and PostgreSQL use SI for enforcing consistency in the database by default [26].

Similarly, database replication research has been focusing on SI and its variants, such as generalized SI, strong SI and weak SI [60]. Some variants are (i) *Weak SI* i.e., write transactions are asynchronously executed on the Replicas and read transactions are sent to any Replicas regardless of their consistency; (ii) *Replicated SI with Primary Copy (RSI-PC)* i.e., Write transactions are asynchronously executed on the Replicas and read transactions are sent to any Replicas that is fully consistent with the Primary DB (but waits if none is available). RSI-PC is implemented by the middleware Ganymed [77]; (iii) *One-copy Serializability (ISR)* i.e., write transactions are synchronously executed on all Replicas and read transactions are sent to any Replica. ISR is the default consistency level of the middleware C-JDBC [27].

However, in any networked shared-data system there is a fundamental trade-off between consistency, availability, and partition tolerance. This is captured by Brewer's CAP theorem, which says it is impossible to have a distributed system which support Consistency (C), Availability (A), and Partition tolerance (P) simultaneously; you can achieve only two of the three. Consistency means everyone gets the same answer; Availability means clients have ongoing access (even if there is partial system failure); and Partition tolerance means correct operation, even if nodes within the system are cut off from the network and unable to communicate. Distributed RDBMS typically CA systems, it supports consistency and availability but not partition tolerance. In simple terms it means, horizontally scaling a RDBMS by adding more machines will not give a guarantee against failure during network outage. *NoSQL* being non-relational, distributed, open-source and horizontally scalable databases typically provide the system designer the ability to trade-off between these choices. Most NoSQL systems implement AP that is also known as Eventually Consistent (EC), because, AP properties do not guarantee immediate consistency. EC requires convergence of replicas, i.e., in the absence of updates and failures

the system converges towards a consistent state. Updates may be reordered in any way possible and a consistent state is simply defined as all replicas being identical. EC is very vague in terms of concrete guarantees but is very popular for web-based services [17].

## 2.2 Scalability

The recent explosion of data has led to the development of innovative systems for large-scale data processing [13]. This large-scale capability requires a highly flexible architecture that easily and quickly adds server nodes to an existing cluster. A scaled distributed database can always be made cost optimal by adjusting the number of hardware resources and the workload. Hardware resources (i.e., Compute and Memory) are adjusted *horizontally* or *vertically*. Vertical scale up is to increase overall system capacity by increasing the resources within existing individual nodes. However, horizontal scale out is to increase overall application capacity by adding nodes. The combination of horizontally and vertically scale up is called *Diagonal* [86]. The current scaled distributed databases inherit their architectures design from the multiprocessor high transaction systems [32]. These systems are classified as:

- **Shared memory**, i.e., when multiple processors share a common central memory. Shared memory is a tightly coupled architecture in which multiple processors within a single system share system memory. This architecture provides high-speed data access for a limited number of processors, but it is not scalable beyond approximately 64 processors when the interconnection network becomes a bottleneck.
- **Shared disk**, i.e., multiple processors each with a private memory share a common collection of disks. Shared disk is a loosely-coupled architecture optimized for applications that are inherently centralized and require high availability and performance. Each processor can access all disks directly, but each has its own private memory.
- **Share all**, i.e., multiple processors share a common central memory and a common collection of disks.

- **Shared nothing**, i.e., neither memory nor peripheral storage is shared among processors. Shared nothing, often known as massively parallel processing (MPP), is a multiple processor architecture in which each processor is part of a complete system, with its own memory and disk storage i.e., a separate nodes.

The above architectures resulted in three Distributed Databases (i) The traditional RDBMS cluster; (ii) The NoSQL data stores and (iii) The NewSQL cluster. Each of them are designed to solve different types of problems, and as a result, their scaling characteristics are different. Beneath, we analyzed these Distributed Databases taking into account their scaling characteristics as well as which workloads are best suited for each system.

**Traditional Relational Database Cluster:** Relational Databases (RDBMS) store tuples. Unlike the other data stores, relational DBMSs have a complete pre-defined schema, offer a SQL interface, and support ACID transactions. Traditionally, RDBMSs have not achieved the scalability performance offered by NoSQL and NewSQL data stores. Traditional RDBMS achieved horizontal scalability based on two architectures, the share nothing architecture and the share all architecture (Their node members can be vertically scaled). The current RDBMS using the share nothing architecture can horizontally scale using one of the following solutions:

- **Master-Slave Replication:** The system's replication is achieved using WAL (Write-Ahead Logging). WAL writes all transactions' modifications in a log file. At that point, WAL's logs are applied serially on slaves nodes [76]. This solution used to improve read performance, where read-only content is accessed on the slave nodes and updates are sent to the master.
- **Middleware Replication:** A middleware layer exists, between the database and the application. The middleware layer is responsible for providing transactions routing, data consistency and Database connection balancing. Examples of such solutions are C-JDBC, Middle-R and DBFarm [25, 74, 79].
- **Multi-Master Replication:** Transactions' writes and reads are distributed to more than one master. Masters co-operate to provide data consistency. MS-SQL[6],



MySQL 6.3+, and the PostgreSQL using third party solutions such as Bucardo, hubyrep, pgcluster and postgresXC [4] are some examples of this type of replication.

- **Master-Slave Replication with Sharding:** Data distribution is similar to Master-Slave Replication i.e., every slave has a copy of the Master Database. The sharding option can be applied for large tables i.e., it allows table data to be partitioned to all cluster nodes (Master and Slaves) based on some hash key. This technique is supported from MS-SQL, MySQL and PostgreSQL.

The Multi-Instance share-all architecture is a mature architecture which has been built over the last thirty years with full ACID compliance. It provides high availability and scalability and also provides multi-instance query parallelism, data partitioning and a plethora of RDBMS features including backups, security, and encryption. The RDBMSs that use share all architecture are horizontally scaled using a Multi-Instance, share disk architecture. Oracle RAC and IBM DB2 Pure-Scale[70, 52] are pioneers in this area. In an IBM DB2 pureScale cluster, each database member has direct memory-based access to the centralized locking and caching services of the PowerHA pureScale server. The IBM PowerHA pureScale server provides centralized lock management services, a centralized global cache for data pages and more. The Oracle RAC, based on cache fusion technology[58], allows running of multiple database instances on different servers in the cluster against a shared set of data files. The database spans multiple hardware systems and appears as a single unified database to the application. However, this solution also has its downsides:

- Horizontal or vertical scaling involves very high licensing costs.
- The system is always overprovisioned, since licensing fees are prepaid for the maximum number of database nodes, regardless of whether a node is idle and will start on demand.
- There is a limitation on the number of nodes per cluster, i.e., 128 nodes per cluster.

- To achieve high performance with this technology, expensive hardware such as high-speed networking and expensive SAN storage, is needed.

A Middleware and Master-Slave replication provides a scalable, low cost, and high performance solution for read intensive workloads. Also, it supports full SQL features and is associated with a plethora of in-core features. However, this solution presents some drawbacks:

- Transaction's properties support ACID with certain relaxations.
- All Writes transactions must be run on Master. Thus, the master node can be a performance bottleneck, since it cannot be scaled horizontally, but only vertically [84].
- The data are cloned to all cluster nodes, i.e., each cluster node has redundant data.

A Multi-Master share nothing replication approach supports ACID with certain relaxations. It provides high availability, both for read and write requests. In addition, it supports full SQL features and is associated with a plethora of in-core features. However, this solution also presents some drawbacks:

- This technology does not support enterprise applications. It needs to strengthen its functionality in order to support more enterprise applications requirements.
- This technology has limited scalability due to limitations on the maximum number of Masters per cluster.
- The data are cloned to all cluster nodes, i.e., each cluster node has redundant data.
- As more masters join the cluster to attend to the increasing demand, the overhead necessary to keep consistency induces delays on requests.

Sharding solutions, provide low cost, high availability, linear scalability and high performance for local transactions. The data are distributed to all nodes and transaction consistency is relaxing to AP [21]. However, this solution has several pitfalls [23, 84, 95]:

- It is expensive when applied to global use reads and updates due to the networking messaging between the nodes. To ensure linear scaling, the scope of a transaction is usually limited to a single node.
- It cannot be used across the board and is applicable only to specific applications.
- Sharding needs extra implementation for shard balancing and data migration applications when shards are added or removed from the system.
- It does not support full SQL.

**NoSQL data stores:** Applications' needs, for storing large amounts of data, move data stores technology towards NoSQL systems. NoSQLs are extremely scalable but generally do not provide ACID transactional properties. Rather, they are eventually consistent. The nature of the NoSQL's applications are generally limited to single nodes, thus, these systems are flexible to scale. NoSQL data stores are mainly classified into three categories [23]:

- *Key-value Stores:* These systems store values and an index to find them, based on a programmer-defined key. Examples of these data stores are Dynamo, FoundationDB, Voldemort, Riak, Redis, Scalaris, Tokyo, cabinet, Memcached, Membrain and Membase.
- *Document Stores:* Document stores are very similar to key-value stores, where documents (data) are stored based on programmer-defined keys and the system is aware of the (arbitrary) document structure (typically JSON). Clusterpoint, MarkLogic, SimpleDB, CouchDB, MongoDB and Terrastore are some examples.
- *Extensible Record Stores:* Their basic data model is rows and columns, and their approach for scalability involves splitting both rows and columns over multiple nodes. Rows are split across nodes through sharding on the primary key. They typically split by range rather than a hash function. Yahoo PNUTS, Bigtable, Cassandra, Accumulo, HyperTable and HBase are some example of Extensible Record Stores.

Despite the significant benefits offered by NoSQL data stores, including ACID properties and SQL features (both properties are offered for some engines), the current technologies are not suitable for the vast majority of relational data. For example, NoSQL is not meant for transactional applications. Also, NoSQL is not a specific type of database or programming interface. The design and query languages of NoSQL databases vary widely between different NoSQL products. This lack of standardization requires extensive application rewrites.

**NewSQL Data Stores:** NewSQL is a new breed of DBMSs, which are divided into two categories: (i) The extended NoSQL systems with ACID and SQL Features and (ii) the completely new database platforms designed to be high scaled databases supporting ACID and SQL Features.

Some examples of the former category are the Google Spanner [35], HyperDex [43] and Splice Machine. Google Spanner is a NoSQL data store which provides all NoSQLs advantages, while offering ACID properties. It is widely used for Google's applications such Megastore, Gmail, Picasa, AppEngine, etc. Google Spanner uses the google file system (GFS) and the Big-Table as a data store [35, 61]. HyperDex is a distributed key-value store that provides a new search primitive for retrieving objects by secondary attributes. Other similar products, like MongoDB and Cassandra, allow an object to be retrieved using only the key under which it was stored. HyperDex achieves this extended functionality by organizing its data using a novel technique called hyper-space hashing. Hyper-space hashing maps objects to servers to enable efficient object insertion and retrieval. Splice Machine extends Hadoop technology to support the relational technology. Splice Machine achieved this by integrating the Apache Derby DBMS [12] in each node on top of H-base[1] and hadoop [2]. Splice Machine utilizes the proven auto sharding capability in HBase to provide massive scalability across commodity hardware, even up to dozens of petabytes. Splice Machine provides ACID transactions across multiple rows and tables to enable real-time updates without data loss or corruption.

However, the rest NewSQL systems are completely new database platforms, handling mainly all types of SQL's queries and they are designed to operate in a distributed cluster

of shared-nothing nodes, in which each node owns all or a subset of the data. Instead of Sharding, NewSQL use intelligent techniques and sometimes specific hardware (i.e., fast disks) and high memory to automatically split and distribute data across nodes. Uniform data distribution is maintained as nodes are added, removed, or when data is inserted unevenly. Storage Engines send the execution of transactions and queries to the nodes that contain the needed data. SQL queries are split into query fragments and sent to the nodes that own the data. NewSQL are drop-in replacements for MySQL and are designed to overcome MySQL scalability issues. They extend MySQL in order to offer ACID guarantees, intelligent indexing for query improvements, online schema modifications, fault-tolerance features for high availability within a cluster, parallel backup and parallel replication among clusters for disaster recovery. Clustrix [30], InnoDB [5], TokuDB [100], etc., are the dominant products in this category. The Clustrix database uses the query to data approach i.e., SQL queries are split into query fragments and sent to the nodes that own the data, in order to run in parallel. This enables Clustrix to scale horizontally as additional nodes are added. InnoDB is a MySQL's storage engine that balances high reliability and high performance. Today InnoDB is the default MySQL storage engine, replacing the previous storage engine MyISAM [67] which does not offer ACID [5]. TokuDB for MySQL is an ACID-compliant transactional database that uses a proprietary Fractal Tree storage algorithm to deliver 20x-50x faster indexing versus databases that use a conventional B-tree algorithm. [100]. NuoDB has a distributed architecture which is split into three type of engines: (i) the administrative engine (AE), (ii) the transaction engine (TE) for concurrency control based on MVCC [85, 66], (iii) the storage manager (SM) for transaction Durability. TE accept SQL client connections, parsing and running SQL queries against cached data. However, SM contains the actual data. SMs and TEs communicate with each other over a simple peer-to-peer coordination protocol. These engines can scale separately and handle failures independently. The system can scale out by spreading TE and SM engines to cluster nodes. MemSQL [64] is an in-memory, scalable, shared-nothing architecture database that is queried through an SQL interface. Storing data in memory eliminates the latency that results from reading and writing to disk. It

uses MVCC and lock-free data structures to enable high throughput for large concurrent workloads without sacrificing consistency. The cluster can be scaled out at any time to provide additional computational resources and storage capacity. Sharding is done automatically, and the cluster re-balances data and workload distribution. Because the data is stored in memory, queries run at full speed on clusters built from commodity hardware. VoltDB [107] is similar to MemSQL. VoltDB is a relational database system (RDBMS) for high throughput, which includes automatic sharding across a shared-nothing server cluster, main-memory data architecture, automatic replication and command logging for high availability and durability.

Each database technology uses a different architecture thus has a different scaling approach. In this section, we examine the current scaling options for each technology, by expressing for each technology the conditions of "How", which indicates what replica management and live migration mechanisms are used during scaling-out and "When", which indicates under which conditions the system scales-out or scales-in discussed in section 2.2.1. The study of these two parameter will help us to identify how these technologies can scaled and how easy these technologies can auto-scale as a feature perspective. Note that we do not examine how data is replicated to replicas as well as how data is read or changed by transactions. These issues are analyzed later in the Database replication section 3.

### **2.2.1 How RDBMSs Scale Horizontally**

Scalability in RDBMSs is dependent on the database system architecture, i.e., share all or share nothing.

In a Multi-Master solution (Figure 1), the database can be scaled by adding a replicated Master node. When a new node joins the database, the state of the database must remain consistent. The complexity of this task depends on the consistency level that should be guaranteed by the Multi-Master databases. The replication phase for the majority of the current Multi-Master solutions are done when the systems are offline, i.e., the systems do not have active transactions. The extension of the system can be done by

adding a new Master database (e.g., shutdown the system and clone any consistent Master Database).

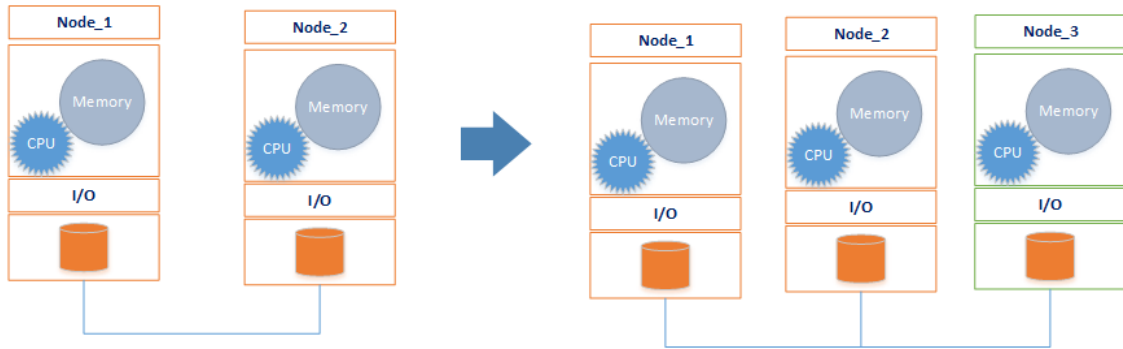


Figure 1: RDBMS Multi-Master Horizontal Scaling

With the Master-Slave solution (Figure 2), the database can be scaled by adding only slave nodes and by adding more hardware resource on Master. The new replicated slave database can join the active database online or offline. The database replication takes place in a similar fashion to the one in Multi-Master architecture. Master-Slave solutions with data sharding can replicate using a complete database rebalancing to all nodes. This task works offline. Another approach is proposed by Soundararajan, et al. [94], which considers online database replication and divides systems' databases into two logical states. In the "READ" state, databases are active and have a consistent view of data. These databases are able to support queries. In the "WRITE" state, the databases are inactive, and they apply committed database's logs from the scheduler in order to be consistent when the system needs more databases.

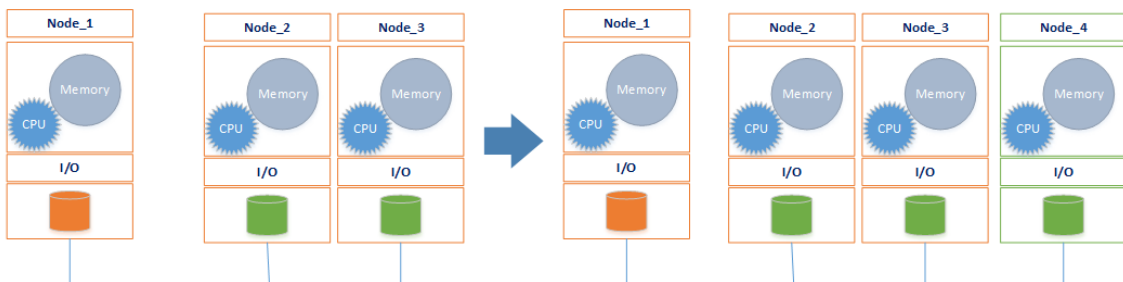


Figure 2: RDBMS Master-Slave Horizontal Scaling

In the share all architecture (Figure 3), the new instance can join the database online. In this case, no data replication or migration is needed, because the data is shared from all database instances and as a result, the instance addition is very fast. However, all the

candidate nodes must have the appropriate software in order to be prepared/registered as a member of the share all cluster during cluster creation.

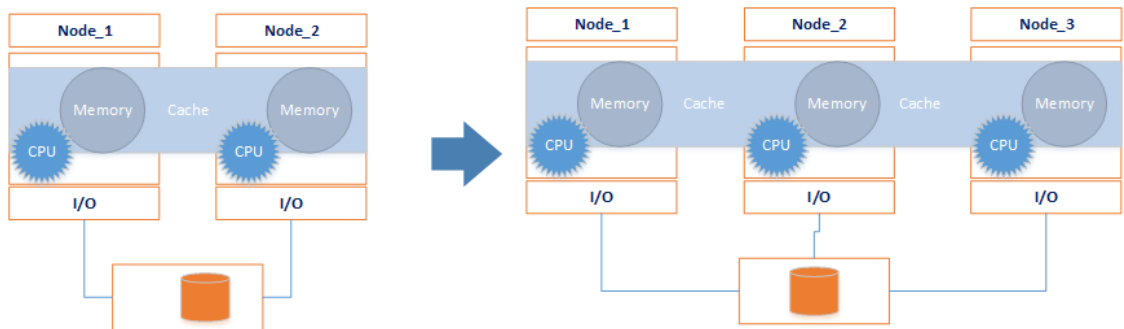


Figure 3: RDBMS Share-All Horizontal Scaling



## 2.3 Elasticity

Elasticity is the system's ability to scale up and scale down capacity based on subscriber workload [50]. In particular, when the load increases the system scales by adding more resources and when demand wanes the system shrinks back by removing unneeded resources. Elasticity is mandatory, if a subscriber's workload defines *workload's quality objectives* (i.e., *Quality Goals*, the strategic quality goal, or combination of goals, that the Service is trying to achieve e.g., *Quality of Metrics X* where *X* can be *Quality of Business* or *Quality of Service* or *Quality of Experience* etc., [68]). Ideally, the system should be able to fulfill workload objectives automatically. However, the challenge of building elastic systems involves adjustment of resources along with load variations without the need for human interventions [105]. Furthermore, Elasticity in Cloud environments is encouraged by the pay-per-use model. On one hand, users do not want to pay for resources they do not currently need, but on the other hand, want to meet rising demands when needed.

According to [50, 53], the *Elasticity's* core design principle conveys three important aspects (i) *scalability*, the ability of the system to sustain workload fluctuations, (ii) *cost efficiency*, acquiring only the required resources by releasing utilized ones, (iii) *time efficiency*, acquiring and releasing resources as soon as a request is made. A system in order to be considered *autonomic scalable i.e., elastic* should support three important conditions. Firstly, its design should be supports scalability. Secondly, the *When* condition should be tightly connected with *Quality of Service* working as driver of when the system will scale-out or scale-in. Thirdly, the design should provide the *How* conditions i.e., what replica management mechanisms should be used in order the system should be able to scale-out or scale-in. Elasticity management could be simple or complex. In simple elasticity management, Scheduling is delegated to a load-balancer that distributes the workload among the VMs provisioned by the elasticity manager. However, In complex elasticity management, the resource provisioning decision arises as a result of the scheduling process. For instance, when the scheduler concludes that the current workload cannot be handled by the available VMs given the impose QoS constraints and the current size of VM job queues, the elasticity manager provisions new VMs from the System in

order to avoid service degradation or penalties and also maintain the end-user satisfaction. A *complex elasticity management solutions* usually consists of multiple components and services such as [22]:

- (i) **Performance and workload monitors:** These are the sensors of the system that are in charge of collecting measures about the key performance and workload indicators. Metrics are always used to evaluate elasticity. There are many kinds of categories of metrics, and many of them are used to evaluate elasticity. Some metrics are of general purpose, and others are specific to elasticity. Example of such metrics are:
  - (a) **response time** e.g., latency, allocation/deallocation, access, idleness, response etc.
  - (b) **throughput** e.g., requisitions/second, Megabytes/second and etc.
  - (c) **utilization** e.g., resource utilization, percentage of CPU utilization and etc.
  - (d) **reliability** e.g., number of violations and etc.
  - (e) **availability** e.g., downtime, uptime and etc.
  - (f) **acquisition** e.g., cost/performance, scalability and etc.
- (ii) **Resource allocator:** This component actuates the resources provisioning actions determined by the elasticity manager.
- (iii) **Load balancer:** It distributes the requests among the instantiated resources. When the elasticity manager is a complex one, load-balancing is replaced by an ASP-customized Cloud-oriented scheduling algorithm.
- (iv) **Elasticity manager:** It plays the central role of compiling the information received from the sensors, reasoning about this input using Service defined policies, and deciding which actions to make. This decision is then sent to the resource allocator that, using the API of the Service, executes the provisioning actions. When new VMs are provisioned, the load-balancer distributes the workload among them.

**Related Work on Elasticity** The work [45] classifies the state-of-the-art elasticity mechanisms. The work defined that the existing automatic policy solutions can be *reactive* or *predictive*, with the ultimate goal of combining various performance objectives such as *performance, cost, energy or increase infrastructure capacity*. The reactive solutions, are based in a *rule-condition-action* mechanisms, however, the predictive solutions uses heuristics and mathematical/analytical techniques. A combination of predictive and reactive approach is also employed to handle the prediction inaccuracy and workloads' oscillations [103]. The work [92] provides a cost-aware elasticity. It computes both a cost-optimized configuration for the desired capacity as well as a plan for transition the application from its current setup to its new configuration. It formulates the provisioning problem as an integer linear program (ILP). A *workload forecaster* forecasts *elapsed time* using an open-source R statistical package that knows the workload in advance. Another work [88, 89] predict workload for feature time periods using a second order auto-regressive moving average method (ARMA) and then identify resource requirements based on the predicted workload using a look-ahead optimization. CloudScale implemented on top of Xen, employs online resource demand prediction and prediction error handling to achieve adaptive resource allocation without assuming any prior knowledge about the applications. The prediction model first employs a fast fourier transform (FFT) to identify repeating patterns called signatures. If a signature is discovered, the prediction model uses it to estimate future resources demand. Otherwise, the prediction model employs a discrete-time Markov chain to predict the resource demand in the near future. The monitoring resources metrics include CPU consumption, memory allocation, network traffic and disk I/O statistics [93].

## 2.4 Discussion

Traditional relational DBMS clusters are scalable, while offering strong consistencies guarantees only in the "Multi-Instance Share All" architecture. The lack of autonomic scaling, the over-provisioned architecture and the high costs of licensing(during scale-out) are important constraints to support the scalability of enterprise workloads. On the

other hand, NewSQL systems achieve horizontal scalability, while offering ACID transactions and SQL, but require expensive hardware (e.g., fast-disks and high memory) and are designed especially for analytical applications. Also, despite the significant benefits offered by NoSQL data stores, including ACID properties, the current technologies are not suitable for the vast majority of relational data. For example, NoSQL is not meant for transactional applications and would require extensive application rewrites. As a result, these weakness imposes the need of offering workload scalability to existing databases without the need for any modifications to either the database engines or the applications. This can be done by the addition of a new middleware sitting between application and replicas that will act as Transaction Manager (TM) or sometimes as *router*, forwarding each application's operations directly to the replicas. The replicas can then execute the operations locally and return the results to the application (e.g., reads can be carried out via a data processor (DP) existing in each Replica). As Amdahl's law implies, workload scalability can be achieved if the problem (in our case the workload) is perfectly parallel, i.e., embarrassingly parallel. Adding a Middleware in the game, there are multiple choices and challenges for how to intercept client queries and implement replication across multiple nodes. For example, a Middleware is typically used to improve either read performance or write performance, while improving both read and write performance simultaneously is a more challenging task. As a result, the new added middleware should have the mechanisms for routing transactions in such a way as to run in parallel (i.e., distributed to all cluster's nodes) while ensuring strong consistency.

### 3 Database Replication for Distributed Databases

Database replication is the process of creating and maintaining multiple instances of the same database. Figure 4 shows a replication execution model. The users execute read and write operations on a *logical data item* and the *replica control protocol* is responsible for mapping these operations to reads and writes on the replicas. As a result, the system behaves as if there is a single copy of each data item, which refers to *one-copy equivalence*.

Distributed databases may be fully or partially replicated. Transactions that access only non-replicated data items are *local transactions*, while, Transactions that access replicated data items have to be executed at multiple sites and they are called *global transactions*.

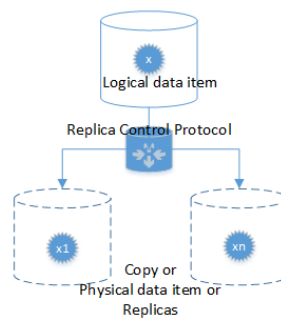


Figure 4: Replication Execution Model

#### 3.1 Concurrency Control for Replicated Databases

Concurrency control is the activity of coordinating concurrent accesses to a database in a multiuser database management system (DBMS). Concurrency control permits users to access a database in a multi-programmed fashion while preserving the illusion that each user is executing alone on a dedicated system [20]. Mutual consistency criteria for replicated databases can either be strong or weak. *Strong* mutual consistency criteria require that all copies of a data item have the same value at the end of the execution of an update transaction. *Weak* mutual consistency criteria do not require the values of replicas of a data item to be identical when an update transaction terminates. What is required is that, if the update activity ceases for some time, the values eventually become identical.

This is commonly referred to as eventual consistency, which refers to the fact that replica values may diverge, but will eventually converge.

A replicated database is said to be in *mutually consistent state* if all replicas of each data item have identical values. Consistency is the criterion that differentiates the current solutions: some ensure that replicas are mutually consistent when an update transaction commits (thus, they are usually called strong consistency criteria), while others take a more relaxed approach (and are referred to as weak consistency criteria).

### 3.1.1 Serializability

*Serializability* is the most widely accepted correctness criterion for concurrency control algorithms. If the concurrent execution of transactions leaves the database in a state that can be achieved by their serial execution in some order, problems such as lost updates will be resolved. A *history*  $R$  (also called a schedule) is defined over a set of transactions  $T = (T_1, T_2, \dots, T_n)$  and specifies an interleaved order of execution of these transactions' operations. A history  $H$  is said to be serializable if and only if it is conflict equivalent to a serial history. In other words, serializability roughly corresponds to consistency such that (i) a transaction  $T_1$  does not overwrite dirty data of other transactions; (ii)  $T_1$  does not commit any writes until it completes all its writes i.e., until the end of transaction; (iii)  $T_1$  does not read dirty data from other transactions; (iv) Other transactions do not dirty any data read by  $T_1$  before  $T_1$  complete. As a result, the primary function of a concurrency controller is to generate a serializable history for the execution of pending transactions. The issue, then, is to devise algorithms that are guaranteed to generate only serializable histories. Consider two sites (A and B), and one data item ( $x$ ) that is replicated at both sites ( $x_A$  and  $x_B$ ). Further consider the following two transactions:  $T_1$ :  $Read_{(x)}, Write_{(x+5)}, Commit$ ; and  $T_2$ :  $Read_{(x)}, Write_{(x*5)}, Commit$ . Assume that the following two *local histories* (the history of transaction execution at each site is called a local history) are generated at the two sites  $H_A = \{(R_1(x_A); W_1(x_A); C_1; R_2(x_A); W_2(x_A); C_2)\}$  and  $H_B = \{R_2(x_B); W_2(x_B); C_2; R_1(x_B); W_1(x_B); C_1\}$ . It is obvious, that although both of these histories are serial, they serialize  $T_1$  and  $T_2$  in reverse order. As a result, the mutual

consistency is violated and both databases are mutually inconsistent. Given the above observation, the transaction consistency is extended in replicated databases to define *one-copy serializability* (1SR) that states the effects of transactions on replicated data items should be the same as if they had been performed one at-a-time on a single set of data items.

Below are presented some different forms of serializability as they have already evolved into the databases:

*Replication with Serializability (SER)* : Whenever a write set is received, a conflict test checks for read/write conflicts between local transactions and the received write set. If the write set intersects with the read set of a concurrent local transaction, the reading transaction is aborted.

*Replication with Cursor Stability (CS)*: The weak point of the SER protocol is that it aborts read operations when they conflict with writes. The protocol may even lead to starvation of reading transactions if they are continuously aborted. A simple and widely used solution to this problem is cursor stability; this allows the early release of read locks. In this way, read operations will not be affected too much by writes, although the resulting execution may not be serializable.

*Replication with Snapshot Isolation (SI)*: Snapshot isolation effectively separates read and write operations thereby avoiding read/write conflicts entirely. This has the advantage of allowing queries (read-only transactions) to be performed without interfering with updates, however, it can suffer from data corruption with some application programs [9]. In fact, since queries do not need to be aware of replication at all, the replication protocol based on snapshot isolation is only concerned with transactions performing updates. In order to enforce the first committer wins rule, as well as to give appropriate snapshots to the readers, object versions must be labeled with transactions and transactions must be tagged with BoT (beginning of transaction) and EoT timestamps. The BoT timestamp determines which snapshot to access and does not need to be unique. The EoT timestamp indicates which transaction made which changes (created which object versions), and hence, must be unique. For example, Oracle uses a counter of committed transactions

to create timestamps. In a distributed environment, the difficulty is that the timestamps must be consistent at all sites. To achieve this, a sequence numbers is added in write sets (WS) messages. Since write sets are delivered in the same order at all sites the sequence number of a write set is easy to determine, unique, and identical across the system [56].

### **3.2 Replication protocols**

According to [46], a fundamental design decision in designing a replication protocol is the choice of update management, which includes:

1. Where the database updates are first performed.
2. How these updates are propagated to the other replicas.

In terms of where updates are first performed, the techniques can be characterized as *centralized* if they perform updates first on a master copy, versus *distributed* if they allow updates over multiple replicas. A *Centralized* technique is the *Master-slave* replication, a popular technique used to improve read performance. In this scenario, read-only content is accessed on the slave nodes and updates are sent to the master. If the application can tolerate loose consistency, any data can be read at any time from the slaves given a freshness guarantee. As long as the master node can handle all updates, the system can scale linearly by merely adding more slave nodes. However, *Multi-master* replication is a *distributed* technique that allows each replica owning a full copy of the database to serve both read and write requests. The replicated system then behaves as a centralized database, which theoretically does not require any application modifications. Replicas, however, need to synchronize in order to agree on a serializable execution order of transactions, so that each replica executes the update transactions in the same order. Also, concurrent transactions might conflict, leading to aborts and limiting the system's scalability.

In terms of how updates are propagated, the alternatives are handled in two ways [46]:

1. Eager: Updates are applied to all replicas of an object as part of the original transaction.



2. Lazy: One replica is updated by the originating transaction. Updates to other replicas propagate asynchronously, typically as a separate transaction for each node.

The update propagation technique that is used in each a protocol typically determines the mutual consistency criterion i.e., eager protocols enforce strong mutual consistency, while, lazy ones enforce weak.

**Eager Update Propagation:** With Eager Update Propagation all updates are applied to all replicas of an object as part of the original transaction. Consequently, when the update transaction commits, all the copies have the same state. Eager propagation techniques are typically implemented using a 2-Phase-Commit (2PC) protocol. Furthermore, eager propagation may use synchronous propagation of each update by applying it on all the replicas at the same time, or deferred propagation whereby the updates are applied to one replica when they are issued, but their application on the other replicas is batched and deferred to the end of the transaction. Eager techniques typically enforce strong mutual consistency criteria. Since all the replicas are mutually consistent at the end of an update transaction, a subsequent read can read from any copy. Writes have to be applied to all replicas. Thus, protocols that follow eager update propagation are known as read-one/write-all protocols.

1. It typically ensures that mutual consistency is enforced using 1SR. Therefore, there are no transactional inconsistencies among replicas.
2. A transaction can read a local copy of the data item and be certain that an up-to-date value is read. Thus, there is no need to do a remote read.
3. The changes to replicas are done atomically.

Eager disadvantages are the following:

1. The response time performance of the update transaction suffers, since it typically has to participate in a 2PC execution and because the update speed is restricted by the slowest machine.

2. If one of the copies is unavailable, then the transaction cannot terminate since not all the copies need to be updated.

**Lazy Update Propagation:** In lazy update propagation, the transaction does not wait until its updates are applied to all the copies before it commits; rather, the replication starts immediately after commit. The propagation to other copies is done asynchronously from the original transaction. The committed transaction is sent to the replica sites some time after the update transaction commits. The main lazy advantage is that it has lower response times for update transactions, since the update transaction can commit as soon as it has updated one copy. The update management strategies provide a classification mechanism for replication protocols according to when the updates are propagated to copies (eager versus lazy) and where updates are allowed to occur (centralized versus distributed). As a result, from the combination of the above arise four replication protocols:

1. Eager Centralized.
2. Eager Distributed.
3. Lazy Centralized.
4. Lazy Distributed.

### **3.2.1 Eager Replication Protocols**

In the Eager Single Master Replication Protocol (figure 5 as demonstrated in [71] chapter 13), the master side orchestrates the operations on the data item. A strong consistency technique is supported so that updates are applied to all replicas within the context of the update transaction, which is committed using the 2PC protocol or other alternatives. Consequently, once the update transaction completes all replicas have the same values for the updated data items. There are two design parameters that need to be taken into account when managing this protocol:

1. The distribution on Master among data items e.g., a single Master for all data items or different masters for a group of data items.

- The level of Replication transparency that is provided. Replication transparency ensures that replication of databases are hidden from the users. Replication transparency is associated with *concurrency transparency* and *failure transparency*.

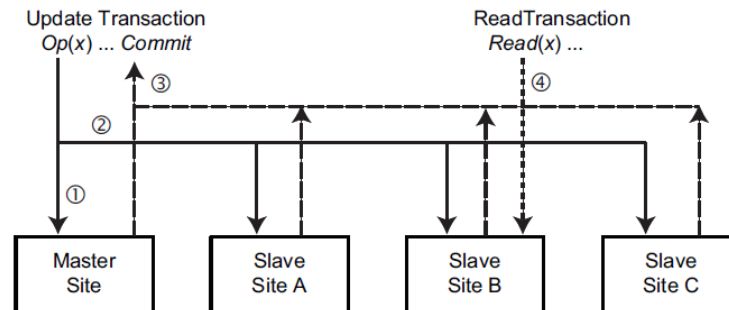


Figure 5: Eager Single Master Replication Protocol Actions (1) A Write is applied on the master copy; (2) Write is then propagated to the other replicas; (3) Updates become permanent at commit time; (4) A Read goes to any slave copy. Figure is adapted from [71] chapter 13.

### 3.2.2 Lazy Replication Protocols

*Lazy centralized replication* algorithms are similar to eager centralized replication ones in that the updates are first applied to a master replica and then propagated to the slaves (see figure 6 as presented in [71], chapter 13). The important difference is that the propagation does not take place within the update transaction, but after the transaction commits as a separate refresh transaction. Consequently, if a slave site performs a Read(x) operation on its local copy, it may read stale (non-fresh) data, since x may have been updated at the master, but the update may not have yet been propagated to the slaves. The first challenge in Lazy centralized replication algorithms is "How can it be ensured that the refresh transactions can be applied at all of the slaves in the same order". A solution to the problem is ordering the transactions according to the transactions' commit time. Because, there is a single master copy for all data items, the ordering can be established by simply using timestamps. The master site would attach a timestamp to each refresh transaction according to the commit order of the actual update transaction, and the slaves would apply the refresh transactions in timestamp order.

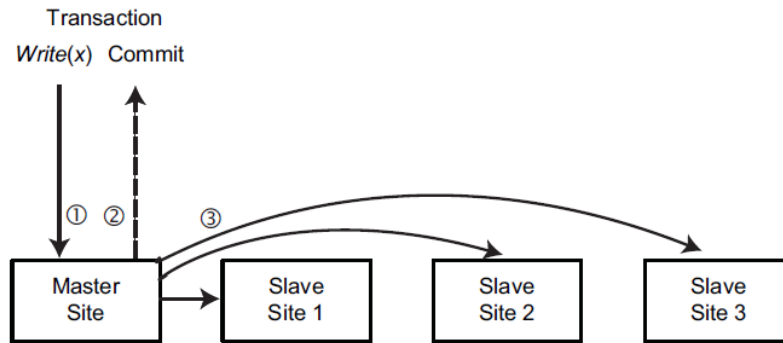


Figure 6: Lazy Single Master Replication Protocol Actions (1) Update is applied on the local replica; (2) Transaction commit makes the updates permanent at the master; (3) Update is propagated to the other replicas in refresh transactions; (4) Transaction 2 reads from local copy. Figure is adapted from [71] chapter 13.

### 3.3 Group Communication

The overhead of replication protocols can be high particularly in terms of message overhead. A critical issue in efficient implementation of these protocols is to reduce the message overhead. A group *communication system* enables a node to multicast a message to all nodes of a group with a delivery guarantee, i.e., the message is eventually delivered to all nodes. In *total ordered* multicast, all messages sent by different nodes are delivered in the same total order at all nodes. Total ordered communication guarantees that all sites receive the write operations in exactly the same order, thereby ensuring identical serialization order at every site. The [56] suggests a group communication based eager distributed protocol. The protocol uses a local processing strategy, where Write operations are carried out on local shadow copies where the transaction is submitted and utilizes total ordered group communication to multicast the set of write operations of the transaction to all the other replica sites. However, [72] provides a Lazy based protocol. The protocol assumes FIFO ordered multicast communication with a bounded delay for communication (call it Max), and assumes that the clocks are loosely synchronized so that they may only be out of sync by up to some time. It further assumes that there is an appropriate transaction management functionality at each site. The result of the replication protocol at each slave is to maintain a “running queue” that holds an ordered list of refresh transactions, which is the input to the transaction manager for local execution. Thus, the protocol ensures

that the orders in the running queues at each slave site where a set of refresh transactions run are the same. Furthermore, the work of [56] proposed four optimizations in order to minimize the overhead of replication protocols. These include: (i) reducing the message overhead (ii) eliminating deadlocks, (iii) optimizations using different levels of isolation and (iv) optimizations using different levels of fault-tolerance.

### 3.4 Middleware Based Replication

*Middleware based* replication uses a middleware layer between the application and the database engines to implement replication. The middleware is sitting between application and replicas and it act as Transaction Manager (TM) or sometimes as a "router", forwarding each application's operations directly to the replicas. The replicas can then execute the operations locally and return the results to the application (e.g., reads can be carried out via a data processor (DP) running on each Replica). When using a Middleware, there are multiple design choices for how to intercept client queries and implement replication across multiple nodes. The solutions are divided into *database kernel* i.e., transaction logs are created by the database's kernel and *non database kernel* i.e., transaction logs are created outside of the database's kernel. In the context of database kernel, internal database's transaction logs are copied *log shipping* from the Primary and applied to the replicas immediately after they are received, or after a short delay. This replication method is called *hot standby* and it offers weak consistency between Primary and Replicas. A strongest consistency it offered via the *streaming replication*, where the Primary and the Replicas have special processes in sync called the *walsender* and *walreceiver* which transmit modified data pages over a network port. This solution is supported by Oracle, PostgreSQL, IBM DB2, Microsoft SQL, and others. However, in the context of non database kernel, the middleware is responsible for gathering and shipping transaction logs to replicas. As a result, at the end of every transaction, transaction logs are organized into *write sets* and applied to the replicas immediately after they are received. This solution is supported by MySQL and Microsoft SQL database as well for various middleware solutions such as Ganymed [78], C-JDBC [28], DBFarm [80] and Middle-R [75]. A Middleware is typ-

ically used to improve either read performance or write performance, while improving both read and write performance simultaneously is a more challenging task.

## 4 Hihooi System Overview

The proposed system called Hihooi is a *replication-based middleware* solution that aims at offering both workload scalability and strong consistency to enterprise databases. As such, Hihooi employs *master-slave replication*, a popular technique used to improve performance for transactional workloads [26]. Transactions are 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. With master-slave replication, all write transactions are sent to the master node, denoted as *Primary DB*, while read transactions are directed to the 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. However, achieving the dual goal of scalability and strong consistency introduces six core challenges addressed by the design choices of Hihooi.

**Challenge 1. Concurrency Control:** *How to efficiently route read transactions to the Extension DBs in a consistent way.*

As a middleware layer between applications and database engines, Hihooi intercepts all incoming transactions and is tasked with routing them either to the Primary DB or to one of the Extension DBs. Since all write transactions are always executed in the Primary DB, Hihooi can safely route there any read queries. This tactic, however, goes against the primary goal of Hihooi to scale performance, which is maximized when the read queries are distributed to the available Extension DBs. The main issue here is that Extension DBs are not always up-to-date with the Primary DB due to the asynchronous propagation of write transactions to the Extension DBs. Hence, Hihooi needs an efficient approach for determining which Extension DBs are consistent with which incoming read queries.

The solution employed by Hihooi consists of three parts. First, for each incoming query  $Q$ , Hihooi extracts the tables, columns, or rows that are potentially modified by  $Q$  (if  $Q$  is an update query) or accessed by  $Q$  (if  $Q$  is a read query). Second, 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. 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. 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. If no consistent Extension DB is found, then the system routes the request to the Primary DB, which is always consistent.

**Challenge 2. *Replica Control:*** *How to efficiently propagate updates from the Primary DB to the Extension DBs in a consistent way.*

To ensure that the read transactions running at some Extension DB see a consistent view of the database, the replica must reflect a transaction-consistent snapshot of the data at the Primary DB; that is, the replica must reflect all data modifications of transactions (up to some transaction) executed at the Primary DB in the serialization order of execution. To retain global system consistency, Hihooi captures the total order of transaction completions in the Primary DB and utilizes *statement replication* (i.e., the write statements are replicated to the Extension DBs), while ensuring that each replica applies writes in the same order. The statement replication takes place *asynchronously* in order to avoid delaying the write transactions executing at the Primary DB. The conventional practice in database replication and hot standby deployments is to apply the writes serially at the slaves, even though the master processes them in parallel [77, 62]. In a heavily loaded production system, however, the lag between the master and a slave node can become significant [26]. Hihooi implements a novel algorithm for applying write transactions in parallel at the slaves, while maintaining strong consistency guarantees.

**Challenge 3. *Backup and Scalability Management:*** *How to efficiently create a new Extension DB from a consistent backup and quickly bring it up-to-date.*

Performing backups is a standard management operation for database systems in order to provide recovery from failures. For a replicated database system, such as Hihooi, it



is even more important because backups can be used to add new Extension DBs into the system *without affecting the performance of the Primary DB or the existing Extension DBs*. Hihooi periodically creates *system checkpoints* which include two important steps: (i) Create a Primary DB consistent Backup called *Seed DB*; (ii) Create a number of Extension DBs by cloning the Seed DB.

During the time needed to create and synchronize the new Extension DB, new write transactions may have been executed at the Primary DB. 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. Instead, as soon as the Extension DB is created and an initial set of write transactions has been executed, it can join the system and start executing read transactions, while concurrently applying the new write transactions. The backup and scalability management procedures are described in Section 8.

## 5 Hihooi System Architecture

Figure 7 depicts the Hihooi architecture, along with the core components and the flow of transactions through the system. As a middleware system, Hihooi is positioned between the applications and the database engines. The custom **Hihooi JDBC/ODBC Drivers** implement the **Hihooi API** and provide database-independent connectivity between the applications and Hihooi. This approach requires the database driver to be replaced in the application but it does not require any other application 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. Currently, Hihooi supports multiple versions of the same engine running concurrently (which is important during database updates) and could support heterogeneous engines in the future. Finally, **HConsole** is an interactive console application that can be used for configuring and managing Hihooi, including adding and removing replicas, creating checkpoints, and executing queries.

The **Transaction Manager** is responsible for intercepting all queries and sorting them into write and read transactions. The write transactions are executed on the **Primary DB**, while the read transactions are load balanced to consistent **Extension DBs**. Apart from managing the client sessions, the Transaction Manager oversees the available Extension DBs and keeps track of which write transactions they have applied. Once a write transaction completes (either via `commit` or `rollback`), the transaction's statements are pushed into the **Transactions Buffer**, which is distributedly stored in memory using Memcached [44]. 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 two Hihooi services; the Extractor and the Delivery Agent. The **Extractor** is responsible for fetching the new write transactions from the Transactions Buffer and applying them to the local database. The Extractors implement a novel algorithm (discussed in Section 6) for executing the transactions in parallel, while respecting the order imposed by the transaction commit timestamps on the Primary DB.

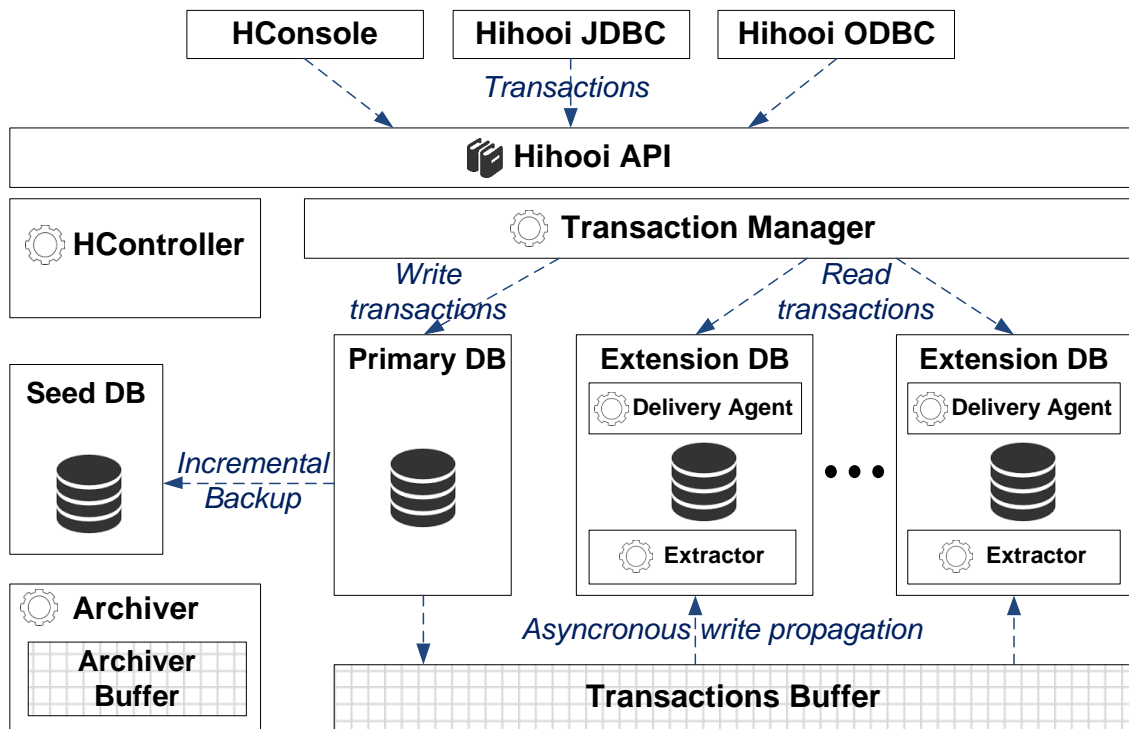


Figure 7: Hihooi Architecture

The **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. When adding new Extension DB into the system, the new Extension DB will be available on the system having one of the following states: (i) **UP**: In this state the Extension DB is started but its replication option is OFF and it is not available for serving queries. (ii) **ACTIVE**: In this state the Extension DB is started and its replication option is ON and it is available for serving queries. (iii) **RECOVERY**: In this state the Extension DB is started and its replication option is ON but it is not available for serving queries. (iv) **SUSPENDED**: In this state the Extension DB is started and its replication option is ON and it is not available for serving queries. (v) **DOWN**: In this state the Extension DB is stopped.

The **Archiver** has a dual role in Hihooi. First, it is responsible for initiating the incremental backups for creating the **Seed DB** based on the Primary 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. Second, 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** in order to keep the memory usage of the Transactions Buffer bounded. A new Extension DB is initialized using the Seed DB, followed by the application of the appropriate write transactions from the Archiver Buffer. Next, the Extension DB notifies the Transaction Manager that it can start serving read queries, while it starts applying the latest changes from the Transactions Buffer. Finally, all system management operations, such as adding and removing Extension DBs, are coordinated by the **HController**.

## 6 Database Replication with Hihooi

Hihooi intercepts and redirects all incoming write transactions (i.e., transactions that modify the database) to the Primary DB for execution. As soon as a transaction completes on the Primary DB, it must be propagated and executed on all Extension DBs, while preserving the completion order from the Primary DB. Before explaining our statement propagation and replication procedure in Section 6.3, we first introduce the notion of *transaction read/write sets* in Section 6.1. Finally, we discuss the benefits and practical considerations of our approach in Section ??.

### 6.1 Transaction Read/Write Sets

Transactions are naturally divided into *single* and *multi-statement*, depending on the number of SQL statements included in the transaction. In most database engines, each SQL statement is considered to be a single transaction by default, and gets committed automatically when it completes execution. Multi-statement transactions are either started with their first statement when automatic commit is disabled (and must be committed manually), or enclosed between specific keywords (e.g., `begin atomic ... end`). Hihooi follows the same conventions. For ease of presentation, we discuss single transactions first, while multi-statement ones are presented if special handling is needed.

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. For example, transactions  $W_1$ ,  $W_2$ , and  $W_3$  shown in Table 1 modify the respective tables  $R$ ,  $S$ , and  $R$ ; 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 6.3) and how to route read transactions efficiently (see Section 7). Suppose  $W_1$ – $W_3$  are executed on the Primary DB and committed in that order. In general, we wish to execute  $W_1$ – $W_3$  on the Extension DBs in the same order to ensure the consistency of the replicas. In this scenario, since  $W_1$  and  $W_2$  modify two different tables, we can execute them in parallel and let them commit in reverse order.  $W_3$ , on the other hand, must execute after

Table 1: Example write transactions on tables  $R(A1^*, A2, A3, A4)$  and  $S(B1^*, B2, B3, B4, B5)$  along with corresponding write sets, read sets, affecting classes, and transaction state identifiers (TSIDs)

TX	SQL Statement	Write Sets			Read Sets			Affect. Class
		Tab.	Col.	Row	Tab.	Col.	Row	
$W_1$	UPDATE R SET A2 = ?, A3 = ? WHERE A1 = 100	R	A2,A3	A1 = 100	R	A1	A1 = 100	RAS
$W_2$	UPDATE S SET B2 = ? WHERE B5 > ?	S	B2		S	B5		CAS
$W_3$	UPDATE R SET A3 = ?, A4 = ? WHERE A2 < ?	R	A3,A4		R	A2		CAS
$W_4$	DELETE FROM R WHERE A1 = 120	R	*	A1 = 120	R	A1	A1 = 120	RAS
$W_5$	UPDATE S SET B4 = ? WHERE B5 < ?	S	B4		S	B5		CAS

the completion of  $W_1$  (as it modifies the same table  $R$ ) in order to preserve consistency.

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. The **Column Read/Write Sets** of a transaction  $T$  denote the columns read/written by  $T$ . Consider transaction  $W_2$  from Table 1.  $W_2$  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 (PK) or a unique key. For instance, transaction  $W_1$  (see Table 1) 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 simple to identify (using a basic query parser) and efficient to compare against each other. The alternative would require reasoning with complex query-level semantics, whose overhead

Table 2: Definitions of read/write sets for relational algebra operations. The read sets of write operations equal the corresponding read sets of expression  $E$

Operation	Notation	Read Sets		
		Table	Column	Row
Select	$\sigma_p(R)$	$R$	$A_i \in p$	$(PK = ?) \in p$
Project	$\Pi_{A_i}(R)$	$R$	$A_i$	
Union	$R \cup S$	$R, S$	$R.*, S.*$	
Set Difference	$R - S$	$R, S$	$R.*, S.*$	
Cartesian Pr.	$R \times S$	$R, S$	$R.*, S.*$	
Aggregation	$G_j \mathcal{G}_{F_i(A_i)}(R)$	$R$	$G_j, A_i$	
Operation	Notation	Write Sets		
		Table	Column	Row
Insert (tuple)	$R \leftarrow R \cup t$	$R$	$R.*$	$(PK = ?) \in t$
Insert	$R \leftarrow R \cup E$	$R$	$R.*$	
Delete	$R \leftarrow R - E$	$R$	$R.*$	read set of $E$
Update	$R \leftarrow \Pi_{A'_i}(E)$	$R$	$A'_i$ if $A'_i \neq R.A_i$	read set of $E$

Symbols:  $R, S$  = tables;  $p$  = predicate;  $A_i$  = attribute;  $F_i$  = function;  $G_j$  = group by attribute;  $t$  = tuple;  $E$  = relational algebra expression

Table 3: Example read transactions on tables  $R(A1^*, A2, A3, A4)$  and  $S(B1^*, B2, B3, B4, B5)$  along with the corresponding read sets, affecting classes, and consistent transaction state identifiers (TSIDs)

TX	SQL Statement	Read Sets			Affect. Class
		Tab.	Col.	Row	
$R_1$	SELECT * FROM R WHERE A2 > ?	$R$	*		TAS
$R_2$	SELECT A3, A4 FROM R WHERE A1 = 100	$R$	$A1, A3, A4$	$A1 = 100$	RAS
$R_3$	SELECT B2, B3 FROM S WHERE B5 < ?	$S$	$B2, B3, B5$		CAS
$R_4$	SELECT A1, B2, B3 FROM R JOIN S ON A1 = B2	$R$	$A1$		CAS
		$S$	$B2, B3$		

would potentially outweigh any of the performance benefits of concurrent execution.

Table 2 formalizes the creation of read/write sets based on fundamental relational algebra operations. All read operations (i.e., select, project, union, set difference, Cartesian product, aggregation) result in table and column read sets that contain the accessed tables and columns, respectively. A select operation with a conjunctive equality predicate on the primary or unique key has a non-empty row read set. Insert, delete, and update operations have both read and write sets. The column write sets include all table columns for insert and delete operations, but only the modified columns for update operations. The row write sets refer to the rows that are modified based on a primary or unique key (if applicable). Finally, the read sets for write operations are based on the items accessed by their involved expressions. Tables 1 and 3 list several write and read SQL statements along with their corresponding write and read sets.

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  $R$ .

These class definitions will be utilized by our algorithms presented later in Sections 6.3 and 7. Tables 1 and 3 also include the affecting classes for each example transaction.

All aforementioned definitions are easily extended to a multi-statement transaction  $T_m$ . The table read set of  $T_m$  is simply the union of all table read sets of the individual SQL statements in  $T_m$ . The same applies for the table/column/row read/write sets of  $T_m$ . As for the affecting class of  $T_m$ , the following rules apply: (i) if all statements in  $T_m$  are “RAS” for table  $R$  then  $T_m$  is a “RAS” for  $R$ ; (ii) if all statements in  $T_m$  are “CAS” for  $R$  then  $T_m$  is a “CAS” for  $R$ ; (iii) otherwise,  $T_m$  is a “TAS” for  $R$ . Finally, DDL statements are handled as “TAS” write statements.



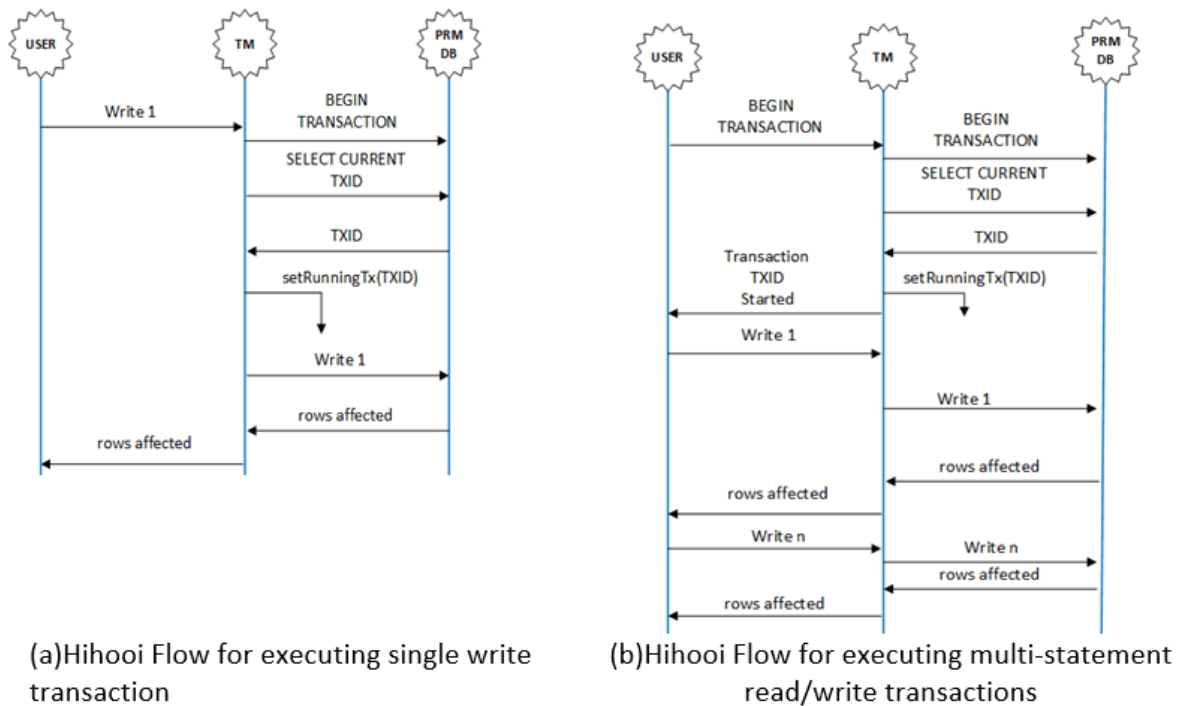


Figure 8: Hihooi processing flow for executing single write and multi-write transactions

## 6.2 Hihooi Commit/Rollback Phase

Figure 8(a) demonstrate the commit/rollback flow for a single write transactions. In the flow three actors are participating: the user who executes transactions, the Transaction Manager and the Primary DB. Furthermore, the Transaction Manager maintains a memory structure called Transactions State (TXState). The TXstate contains the Transactions' state into four fields the TXID, TSID, Commit Timestamp and STATUS. The system manages all the write statements as Single Write Transactions if are not started with identifier "Begin Transaction" and automatically committed. When a Transaction Manager receives a new write statement from the User, it begins a new Transaction on Primary DB and retrieved the assigned Transaction's TXID. Then, the current TXID is added (function setRunning(TXID)) in Transaction Manager's TXState with status RUNNING. At that point, the Transaction Manager executes the write on Primary DB and propagates the affected rows to back to the User. Figure 8(b) demonstrate the commit/rollback flow for a multi-statement read/write transactions. The user submits the "Begin Transaction" identifier indicate a Start Transaction. The Transaction manager propagates the Begin Transaction to the Primary DB and retrieves the assigned Transaction's TXID. Then, the

current TXID is added as RUNNING in Transaction Manager's TXstate. At that point, the Transaction Manager informs the user that the Transaction with TXID has started. From that point and after the user can submit one or more write statements. All the writes statements are propagated and executed on Primary DB and the results are send back to the user. Unlike the Single write statement Transactions the commit command is requested by the user.

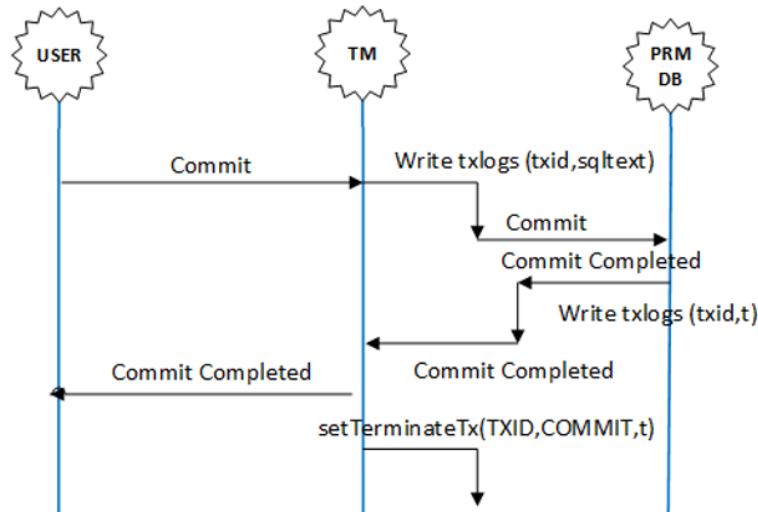


Figure 9: Hihoor Commit/Rollback Flow for single write or multi-write transactions

On commit time, figure 9, either Single write or Multi-statement writes statements, the Transaction's logs (i.e., txid and sql\_text) are written in a file called txlogs before commit and thereupon the Transaction is committed to Primary DB. The return Transaction's commit timestamp is returned back to the Transaction Manager that writes it in txlogs (txid,t). txlogs file is used for Transaction Manager recovery in case of failure. Finally, the Transaction Manager returns a commit completed back to user, and updates TXState, for the corresponding TXID by changing the status to COMMITED associated with the commit timestamp already retrieved by Primary DB(setTerminate(TXID,COMMIT,t)).

### 6.3 Statement Replication Procedure

For each transaction  $T$  intercepted by Hihoor, a *Transaction State* (or *TState*) is built and maintained at the Transaction Manager. A TState contains the following:

1. a TState identifier (*TSID*) that uniquely identifies  $T$ ;

2. the *SQL write statements* of  $T$  in the order of execution;
3. the read/write sets of each statement and the overall  $T$ ;
4. the total execution time of  $T$  on the Primary DB; and
5. the completion operation: `commit` or `rollback` (the replication of failed transactions is explained in Table 4).

**TSID Generation:** TSID is a unique sequential number given to the TState of  $T$  once  $T$  commits or rollbacks on the Primary DB. The purpose of the TSIDs is to capture the order of transaction completions on the Primary DB, which is determined by the *transaction commit timestamps* as recorded by the underlying database system. In order to ensure the correct ordering, the Transaction Manager (TM) performs the following steps after  $T$  completes: (1) the TM obtains the commit timestamp of  $T$  and the commit timestamps of any other concurrent transactions that have issued a commit request but have not received a response yet; (2) the TM issues TSIDs for these transactions in the same order as the commit timestamps. Typically,  $T$ 's timestamp is the lowest and it simply receives the next TSID number. Occasionally though, commit responses are received out of order (up to 2% of the times in our most write-intensive experiments) due to multi-thread scheduling in the TM or network delays. Hence, the procedure above is necessary to ensure that TSIDs are given in the same order induced by the commit timestamps. The TState must be given a TSID before it can be fetched and replayed by the Extension DBs.

**TSID Implementation:** The Transaction Manager has two important daemon processes the Transactions Set Monitor (T-MON) and Transactions Set Archiver (T-ARC) which are responsible for maintaining TSIDs. The T-MON is responsible to generate the sequential TSIDs from the committed transactions and push the Transaction sets into the Transaction Buffer as part of the replication procedure. Specifically, T-MON performs the following: (1) Reading from Transactions State, it generates TSID for those Transactions which have status `COMMITTED` or `ROLLBACK` based on transaction commit timestamp, (2) For Transaction Manager's recovery in case of failure, the TSID:TXID entry is written into disk (tslogs), (3) Push Transaction Sets into Transaction Buffer(if successfully pushed then update the corresponding transaction with status `REPLICATED`). The T-ARC is re-

sponsible to archive the COMMITTED transactions into disk (on successfully write, the corresponding transaction in Transaction List is marked as COMMITTED ARCHIVED). The Transaction Set's archiving is started when the system assigning a TSID to the candidate Transaction Set, otherwise the archiving is waiting. Another, task of T-ARC is to remove Transaction Sets from Transaction Buffer. A Transaction Set is removed from Transaction Buffer if already applied to all active Extension DBs.

**Extension DBs Replication:** On system startup, the Transaction Manager loads the last TXID and TSID from tslogs file (the TSID numbering will continue from this value). Therefore, T-MON is reading from Transactions State and it generates TSID for those Transactions which have status COMMITTED or ROLLBACK based on transaction commit timestamp. The new TSIDs with the corresponding TXID are written into tslogs. Thereupon, T-ARC archives the already replicated Transactions Sets into disk and update Transactions State as ARCHIVED. Then, T-MON pushes the committed Transaction Sets into Transaction Buffer. If the Transaction Sets are successfully pushed into Transaction Buffer, the corresponding entry into Transactions State is marked as REPLICATED. Now, on the Extension DBs' side, on startup, each Extension DB loads the last TSID from its local tslogs file and notifies the Transaction Manager about its TSID, and it starts seeking from Transaction Buffer for a TSID + 1 without knowing at all whether or not there is this TSID. However, if the Extension DB's TSID is less than Transaction Manager's TSID, then, the Transaction Manager either replicated the missing Transaction Sets or invoke the Archiver to reload from archives files all the missing Transaction Sets into the Transaction Buffer. When a new TSID is applied on an Extension DB, the Extension DB always notifies the Transaction Manager about the new TSID.

Each Extension DB node hosts an Extractor service, which is responsible for receiving the completed TStates from the Transactions Buffer and executing them on the local Extension DB. The goal of the Extractor is to ensure that the local database replica is consistent with the Primary DB. Executing the transactions from the TStates in the serial order imposed by the TSIDs is a sufficient condition to achieve consistency. However, it is very inefficient and can cause the Extension DBs to fall significantly behind the Primary

DB, especially in times of heavy write load (given the parallel execution at the Primary DB). Hence, it is crucial for the Extractors to execute in parallel as many transactions as possible while maintaining correct consistency semantics.

As it was alluded in Section 6.1, the read/write (R/W) sets of the transactions are the backbone for our parallel execution algorithm. Specifically, the R/W sets of two write transactions can be used to determine whether the transactions affect the same data items in the database, as shown in Algorithm 6.1. If they don't affect the same items, we say they are **independent**. When the table R/W sets of two write transactions are disjoint, they are independent as they modify different tables (lines#2-4). Otherwise, we need to check which columns and/or rows are modified by the two transactions, but only for the commonly modified tables (line#6). If the two transactions both belong to class "CAS" for a table  $t$  (i.e., they affect some columns of  $t$ ) but do not modify the same columns, then they are independent for  $t$  (lines#8-11). Similarly, if the two transactions both belong to class "RAS" for a table  $t$  (i.e., they affect some rows of  $t$ ) but don't modify the same rows, then they are independent for  $t$  (lines#12-15). If either of the above two conditions holds for all common tables, then the transactions are independent and it is safe to execute them concurrently.

One important property of our R/W set definitions is their *cumulative* nature. That is, if we take the union of the R/W sets of multiple statements, we get R/W sets of a multi-statement transaction with the same correct semantics, as explained in Section 6.1. With the same reasoning, we can combine the read/write sets of two or more multi-statement transactions that are running in parallel to build a transaction state that represents all running statements as if they were one bigger multi-statement transaction. For example, suppose transactions  $W_1$  and  $W_2$  from Table 1 are executed together by an Extractor. Then, we can define a *running transaction state* that includes the merged states of  $W_1$  and  $W_2$ . The table write set of this new state would include tables  $R$  and  $S$ , meaning both tables are currently being modified. 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

---

**Algorithm 6.1** Check independence between two write transactions

---

```
1: function AREINDEPENDENT( $ts_1, ts_2$ )
2:   if  $ts_1.table\_w\_set \cap ts_2.table\_rw\_set = \emptyset$  &
3:      $ts_1.table\_rw\_set \cap ts_2.table\_w\_set = \emptyset$  then
4:     return true
5:   end if
6:   for each  $t$  in  $ts_1.table\_rw\_set \cap ts_2.table\_rw\_set$  do
7:     bool independent  $\leftarrow$  false
8:     if  $ts_1.class[t] = CAS$  &  $ts_2.class[t] = CAS$  &
9:        $ts_1.col\_w\_set[t] \cap ts_2.col\_rw\_set[t] = \emptyset$  &
10:       $ts_1.col\_rw\_set[t] \cap ts_2.col\_w\_set[t] = \emptyset$  then
11:       independent  $\leftarrow$  true
12:     else if  $ts_1.class[t] = RAS$  &  $ts_2.class[t] = RAS$  &
13:       $ts_1.row\_w\_set[t] \cap ts_2.row\_rw\_set[t] = \emptyset$  &
14:       $ts_1.row\_rw\_set[t] \cap ts_2.row\_w\_set[t] = \emptyset$  then
15:       independent  $\leftarrow$  true
16:     end if
17:     if independent == false then return false end if
18:   end for
19:   return true
20: end function
```

Notation:  $*\_w\_set$  = (table | column | row) write set;  $*\_rw\_set$  = union of (table | column | row) read and write sets

---

transaction state.

Algorithm 6.2 shows the two functions that constitute the parallel execution algorithm employed by the Extractor. The new transaction states are received by the Extractor in the order imposed by the Primary DB execution. For each new transaction state  $tsNew$ , the Extractor checks if it is independent from both the running and waiting states, i.e., the combined states of the already running and waiting transactions, respectively (lines#5-6). If it is,  $tsNew$  can be executed in parallel with the already running transactions, after its state is merged with the state of the running transactions (lines#7-8). Otherwise,  $tsNew$  must wait in the queue and its state must update the waiting state (lines#10-11). It is important to check  $tsNew$  against the waiting transactions because a conflict indicates that an already waiting transaction will modify a data item that  $tsNew$  will affect, and these changes must occur in order.

Suppose the five write transaction from Table 1 must be executed at an Extension DB in that order. Transactions  $W_1$  and  $W_2$  will execute in parallel as they modify different tables, while  $W_3$  is placed in the wait queue since it conflicts with  $W_1$ . Even though  $W_4$  is

---

**Algorithm 6.2** Parallel transaction execution at Extension DBs

---

```
1: runningState                                ▷ combined state of running transactions
2: waitingState                                ▷ combined state of waiting transactions
3: waitQueue                                   ▷ FIFO queue with waiting transaction states
4: function ONNEWTRANSACTION(tsNew)
5:   if areIndependent(runningState, tsNew) &
6:     areIndependent(waitingState, tsNew) then
7:     runningState.merge(tsNew)
8:     execute(tsNew)
9:   else
10:    waitingState.merge(tsNew)
11:    waitQueue.enqueue(tsNew)
12:   end if
13: end function
14: function ONTRANSACTIONCOMPLETE(tsOld)
15:   runningState.remove(tsOld)
16:   while waitQueue.isNotEmpty() &
17:     areIndependent(runningState, waitQueue.peek()) do
18:     tsRun ← waitQueue.dequeue()
19:     waitingState.remove(tsRun)
20:     runningState.merge(tsRun)
21:     execute(tsRun)
22:   end while
23: end function
```

---

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 a running transaction completes execution, its state is removed from the running state (line#15). Next, the wait queue is iterated, checking if the next waiting transaction is independent from the running transactions (lines#16-17). If it is, its state is moved from the waiting to the running state and then submitted for execution (lines#18-21). In our running example, when  $W_1$  completes execution,  $W_3$  can then execute, followed by  $W_4$ .

Even though we refer to the read/write sets as “sets”, they are internally implemented using *hash tables*, where the key is the data item (i.e., table, column, or row) and the value is a counter to track how many query statements access that data item. In addition, the column and row read/write sets are maintained separately per database table (also using hash tables), to facilitate their direct use in Algorithms 6.1 and 7.1 (presented in Section 7.1). The merging of two transaction states is a straightforward process. For each corresponding read/write set, the underlying hash tables are merged as follows: if the two

hash tables contain the same key, the associated counters are added together; otherwise the entries are simply put in the resulting table. The deletion of a transaction state from a previously merged state involves decreasing the counters kept for each read/write set. If a counter reaches zero, then the corresponding entry is deleted from the hash table. Hence, all operations on read/write sets are very efficient to implement in practice.

Finally, the Extractor is responsible for notifying the Transaction Manager with the latest applied TSID in sequential order without gaps. Suppose the transactions complete in the order  $W_1, W_3, W_2$ . When  $W_1$  completes, its TSID is reported but when  $W_3$  completes nothing is reported. Once  $W_2$  completes, the TSIDs of  $W_2$  and  $W_3$  are reported. Hence, the Transaction Manager is aware of up to which transaction has been replayed on each Extension DB in sequential order. This information is stored in a simple hash table that maps the Extension DBs to their latest applied TSID.

**Transaction State Implementation Details:** The Transaction States—and the contained read/write sets in particular—are crucial for the efficiency of both the parallel replication procedure of write transactions (discussed in Section 6.3) and the routing of read transactions to Extension DBs (recall Sections 7.1 and 7.2). In this section, we describe the implementation of the data structures used to hold the read/write sets as well as the merge and delete methods used in Algorithm 6.2. Even though we refer to the read/write sets as “sets”, they are internally implemented using *hash tables*, where the key is the data item (i.e., table, column, or row) and the value is a counter to track how many query statements access that data item. For example, consider a multi-statement transaction  $T_m$  containing the statements  $W_1$  and  $W_3$  from Table 1, both of which modify table  $R$ . The table write set of  $T_m$  contains the entry  $\{R \rightarrow 2\}$ , indicating that table  $R$  is modified by two statements. The counters are necessary for the correct and efficient implementation of the delete method, described below. In addition, the column and row read/write sets are maintained separately per database table (also using hash tables), to facilitate their direct use in Algorithms 6.1 and 7.1. Consider transaction  $R_4$  in Table 3, which reads columns  $R.A1, S.B2$ , and  $S.B3$ . The column read set of  $R_4$  contains  $\{R \rightarrow \{A1 \rightarrow 1\}, S \rightarrow \{B2 \rightarrow 1, B3 \rightarrow 1\}\}$ . The merging of two transaction states,  $ts_1$  and  $ts_2$ , is a straightforward process. For each



corresponding read/write set, the underlying hash tables are merged as follows: if the two hash tables contain the same key, the associated counters are added together; otherwise the entries are simply put in the resulting table. For example, if the table read set of  $ts_1$  is  $\{R \rightarrow 2\}$  and of  $ts_2$  is  $\{R \rightarrow 1, S \rightarrow 1\}$ , then the merged table read set is  $\{R \rightarrow 3, S \rightarrow 1\}$ . The deletion of a transaction state from a previously merged state involves decreasing the counters kept for each read/write set. If a counter reaches zero, then the corresponding entry is deleted from the hash table. Suppose  $ts_2$  with the table read set  $\{R \rightarrow 1, S \rightarrow 1\}$  is now deleted from the merged state  $\{R \rightarrow 3, S \rightarrow 1\}$ . The resulting table read set is  $\{R \rightarrow 2\}$ . In the absence of counters, the read set would end up empty in the above example, which would be incorrect since  $ts_1$  is still present. The alternative would involve maintaining the individual read/write set of all transactions and recomputing the merged state each time a transaction was deleted from the state; which would not be as efficient as our solution of maintaining simple counters.

Hihooi employs *statement replication* to ensure consistency in the replicas; i.e., it replicates and executes all write statements on the Extension DBs. The alternative approach would be to use *row-based replication*, which entails capturing the modified table rows on the Primary DB and replicating them to the Extension DBs [55, 73, 41]. One method for achieving row-based replication is to integrate the middleware with the underlying database engine for extracting and adding table rows. This limits the ability of the middleware to use different database engines (or even different versions of the same engine) [82]. Another method is to (i) declare triggers on every table for extracting the modifications and (ii) use a complex mechanism for applying the changes to the replicas. A serious drawback here is the performance overhead introduced on the primary database from the multiple trigger executions.

Hihooi avoids the aforementioned limitations of row-based replication by using statement replication. Hence, it is capable of supporting multiple, unmodified database engines as well as preventing any unnecessary overheads at the Primary DB. In addition, update and delete statements that affect multiple rows are very efficient to replicate as only the SQL statements are propagated to the replicas [82]. Finally, capturing statements

Table 4: Summary of Practical issues and resolutions for statement replication

Practical Issue	Resolution
Database sequences, used to generate unique or auto-incremented keys, are non-transactional objects	Failed transactions are also executed on the Extension DBs to increment any sequences consistently. Read transactions that perform sequence operations are treated as write transactions (i.e., executed on the Primary DB and replicated to the Extension DBs).
Time-related macros such as <code>now</code> or <code>current_timestamp</code>	Query rewriting techniques are used to replace a macro with an actual value that will be common to all replicas.
User-defined functions in write transactions	Non-deterministic functions are executed once and their return values are used in the calling statements. Deterministic functions are left as is.
Stored procedures and database triggers	The R/W sets are extracted from deterministic procedures and triggers upon their definition. For deterministic procedures with non-SQL definitions, the DB admin must provide their table R/W sets during system configuration. Non-deterministic procedures/triggers are currently not supported.

is the basis for extracting the R/W sets, which are used to improve the performance of both the replication and the query routing procedures.

## 6.4 Resolutions against Statement Based Replication Challenges

The main practical issues concerning statement replication arise due to *non-deterministic queries*, i.e., queries that may not produce the same result even when executed on the same database state. Statement replication requires that the execution of a write statement has the same effect on the Primary DB as on the Extension DBs. However, an SQL statement could legitimately produce different results on different replicas if, for example, it referenced sequences, or used the current timestamp, or invoked a non-deterministic function (e.g., `RAND`). Hihooi resolves such issues by (i) performing on-the-fly query rewriting before submitting the queries for execution; and (ii) replicating all, even failed, transactions. Table 4 lists the main practical issues along with Hihooi’s resolutions in the present implementation. Currently, Hihooi does not support the small set of non-deterministic procedures and triggers. Below we describe in details the main challenges presented when using Statement Based Replication and their resolutions:

**Challenge 1. *SQL Statement Execution Order:*** *The order of execution for every SQL statement is significant. The system should ensure that all SQL statements will be executed*

*on Extension DBs in the same order as they were executed in the Primary DB.*

**Resolution 1.** *The TSID sequencing mechanism ensures that the system executes transaction's on Extension DB, in the same order as they were executed in Primary DB.*

**Challenge 2. Table columns with auto increment number:** *A table's columns with auto increment numbers. The system should ensure that new rows added in Primary DB will be replicated uniformly in the Extension DBs.*

**Resolution 2.** *The system replays everything done in Primary DB even statement failures. As a result, unsuccessful transactions (e.g., due to a DBMS exception or primary key and unique key violations etc.) are also logged and replicated, in this way, lost sequence's numbers due to failures are also lost in the Extension DBs (e.g., sequences for auto increment columns or sequences used to generate their next value inside a DML Statement).*

**Challenge 3. Table default values:** *A table's columns with default values based on Date-Time functions. The system should ensure that default values assigned to Primary DB will produce the same results on the Extensions DBs.*

**Resolution 3.** *The same resolution as 2.*

**Challenge 4. A table's columns with default values based on User-Defined functions:** *The system should ensure that default values assigned to Primary DB will produce the same results on the Extensions DBs.*

**Resolution 4.** *The same resolution as 2.*

**Challenge 5. Date-Time Functions:** *The system should ensure that Date-Time Functions may used inside a DML statements either as assignment values, or used as predicates, will not produce different values during replication.*

**Resolution 5.** *Database's Date-Time functions can be used inside DML statements to produce new values or they are used as conditions predicates for the WHERE clause. For example, a Date-Time function can store its return value inside an INSERT statement, in order to provide a new value for a DATE-TIME column. Also, the Date-Time functions*

can be used in an `UPDATE` statement in order to store a new value for a column. In addition, they are used as condition predicates in the `UPDATE`'s and `DELETE`'s `WHERE` clause. During statement base replication, Date-Time functions, cannot provide consistent values, for Primary DB and Extension DB columns because these functions are called in at different times during replication. For this reason, the system rewrites all the DML statements that contain Date-Time functions, and completes the candidate DML statements with the exact values. This background process, and all supplementary queries are performed as part of the current transaction, using the session's primary DB connection. The new SQL statement is executed on Primary DB and logged into Transaction Manager in order for it to be replicated on the Extension DBs.

**Challenge 6. User-Defined Functions and Procedures:** *The system should ensure that User-Defined Functions inside DML statements that are either used as assignment values, or are used as predicates, produce the same results on the Extensions DB, as were originally present on the Primary DB.*

**Resolution 6.** *It is possible for User-Defined functions to exhibit nondeterministic behavior, since their context is difficult to control during replication. To ensure data consistency on statement replication, the system rewrites all SQL statements that contain User-Defined functions, and replaces them with their return values. The system using the transaction's primary connection executes the function as a select statement i.e., `SELECT function_Name()` and rewrites the original SQL statement with the function's return values. Unfortunately, all supplementary queries used for a User-Defined function, must be also logged and executed on Extension DBs again, providing an extra overhead. Imagine a User-Defined function that gets the next value of a sequence `S`, by selecting the function, then the sequence `S` is changed, and, this change on sequence `S` must be replicated in Extension DBs. In order to reduce any supplementary overheads against the User-Defined functions, the system gives the user the option, to mark User-Defined functions as deterministic and nondeterministic. All supplementary logs used for deterministic User-Defined functions are not logged and thus, are not replicated to Extension DBs. The default behavior for Service's Database User-Defined functions are nondeterministic.*

**Challenge 7. Database Triggers:** *Database Triggers may show nondeterministic behavior on statement based replication.*

**Resolution 7.** *Database Triggers should also have nondeterministic behavior, and statement based replication cannot be achieved because, of the difficulty in controlling triggers' context. To ensure data consistency on statement replication, the system disables all tables' triggers except of the triggers that are manually marked as deterministic by the user. The default behavior for Service's Database Triggers are nondeterministic.*

## **6.5 Hihooi and Primary DB Consistency Challenges**

A system checkpoint initializes the consistency between the Hihooi and Primary DB. From the checkpoint and after, all Transactions executed on the Primary DB should be replicated on Extension DBs. Even if one transaction is lost or transactions are applied in different order, the consistency between Hihooi and Primary DB will be broken. Furthermore, the Primary DB is vulnerable to external interference, which cannot be controlled by Hihooi, because, Hihooi and Primary DB are two separated entities. As a result, the Hihooi must be able to detect that the state of the Primary DB is not changed outside of the Hihooi context. Following are some known challenges where a consistency check is mandatory:

**Challenge 1. A Primary DB is detached from Hihooi:** *For example, the customer detaches Primary DB from Hihooi in order to load offline data or the customer's needs for system's scalability is on demand.*

**Resolution 1.** *At any time, the Primary DB can be disconnected from Hihooi. However, in the case of reconnection a new checkpoint creation is mandatory in order to reinitialize the consistency between the Primary DB and Hihooi.*

**Challenge 2. External connections with Primary DB:** *When the Hihooi is binding with Primary DB, nothing prevents other applications to connect with the Primary DB and change its contents. This possibility creates more challenges in preserving consistency for the middleware replication solutions.*

**Resolution 2.** *Our resolution to the problem is focused on three approaches:*

**1. Prevent or block external connections from Primary DB:**

- (i) Open the Primary DB to read only transactions and force Hihooi to use only restricted sessions that are able to create write transactions.*
- (ii) Offer a limited number of available Primary DB connections and reserved all connections to Hihooi.*
- (iii) Binding Shutdown and Startup task for Primary DB and Hihooi.*
- (iv) Limit and lock unused Primary DB's users.*
- (v) Limit and block TCP/IP connections to Primary DB from other sides, and open TCP/IP connections only from Hihooi's sides.*

**2. Detect transactions outside Hihooi:** *All transactions that are executed outside of Hihooi cannot be controlled by Hihooi because they cannot be replicated. As a result, the consistency cohesion between Hihooi and Primary DB is broken. These transactions are difficult to prevent, but their actions can be traceable. The first check in order to detect if any transaction commit/rollback outside of Hihooi, is completed during the reconciliation phase that takes place during the Transaction Manager's startup procedure. The Primary DB's latest commit TXID is compared with the latest commit TXID saved in Transaction manager's metadata (recall Hihooi Commit/Rollback phase). If both TXIDs are equal, the Transaction Manager is open normally, otherwise, the system recovery takes place. When the system opens normally, the Transactions Monitor (TX-MON), which is responsible to detect invalid external transactions, takes the appropriate actions such as killing external transactions or stops the replication. The TX-MON selects from the primary DB all the running TXIDs and compares with the running TXID listed in the TXState. All the TXIDs that are not included in the Transactions State are terminated.*

**3. Resolve transactions inconsistencies:** *Hihooi resolves Transaction inconsistencies by creating a new system checkpoint; however, we are concern to find a fine-grained solution as part of future work.*

## 7 Concurrency Control

As explained in Section 6, all 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 up to the latest transaction (which has the largest TSID) that was 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 or be redirected to the Primary DB. The first option introduces latency delays for the read transactions, while the second further 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 read/write 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 (see Section 7.1). Further, Hihooi can perform an even finer-grained load balancing by directing individual read statements from within multi-statement write transactions to Extension DBs. To the best of our knowledge, *Hihooi is the first replication-based middleware system to offer statement-based routing*, while respecting transaction boundaries and maintaining consistency (see Section 7.2).

### 7.1 Transaction-level Load Balancing

The goal of transaction-level load balancing is to direct read transactions to Extension DBs that are consistent with the Primary DB but only with regards to the data each read transaction will access. In order to achieve this efficiently, Hihooi needs quick access to the tables, columns, or rows accessed by the read transactions as well as to which tables, columns, or rows are up-to-date on each Extension DB. The former is achieved using the

---

**Algorithm 7.1** Update the Transaction Manager hash indexes after executing a write transaction

---

```
1: function UPDATEINDEXES(ts)
2:   TIndex.multiPut(ts.table_write_set, ts.TSID)
3:   for each t in ts.table_write_set do
4:     if ts.class[t] = TAS || ts.class[t] = CAS then
5:       CIndex.multiPut(ts.col_write_set[t], ts.TSID)
6:     else if ts.class[t] = RAS then
7:       RIndex.multiPut(ts.row_write_set[t], ts.TSID)
8:     end if
9:   end for
10: end function
```

---

transaction read sets, while the latter using the TSIDs of the completed write transactions and a set of hash indexes maintained by the Transaction Manager. In particular, three hash indexes are used for separately mapping tables, columns, and rows to the latest write transaction that modified them. Hence, the indexes can be used to find the transaction after which the replica is consistent with regards to specific tables, columns, or rows.

Once a write transaction  $T_w$  completes, its write sets are used to update the three indexes, as shown in Algorithm 7.1. All tables referenced in the table write set of  $T_w$  are added into the *Tables Hash Index* (*TIndex*) and mapped to the transaction state identifier (TSID) of  $T_w$  (line#2). This action indicates that the latest transaction to update those tables is  $T_w$ . Next, the modification of the other two indexes depends on the affecting class of  $T_w$  for each table. In particular, if  $T_w$ 's class is "TAS" or "CAS", then all columns in  $T_w$ 's column write set are added into the *Columns Hash Index* (*CIndex*) and mapped to  $T_w$ 's TSID (lines#4-5). Otherwise, all rows in  $T_w$ 's row write set are added into the *Rows Hash Index* (*RIndex*) and mapped to  $T_w$ 's TSID (lines#6-7). Row entries in *RIndex* that have been applied to all replicas are periodically pruned to keep the index size bounded.

Consider the five write transactions of our running example shown in Table 1. After their execution on the Primary DB, the content of the three hash indexes is shown in Table 5. Each entry (of any index) shows the last TSID that modified that particular item. For example, table *S* was last modified by transaction  $W_5$  with TSID=15, while column *S.B2* was last modified by  $W_2$  with TSID=12.

The last step in the transaction-level load balancing is to determine which Extension DBs are consistent for running an incoming read transaction  $T_r$ . Algorithm 7.2 shows



---

**Algorithm 7.2** Find the latest consistent TSID for a read transaction on the Transaction Manager
 

---

```

1: function FINDLATESTCONSISTENTTSID(ts)
2:   tsid = 0
3:   for each t in ts.table_read_set do
4:     if not TIndex.contains(t) then skip iteration
5:     if ts.class[t] = TAS then
6:       tsid = max{tsid, TIndex.lookup(t)}
7:     else if ts.class[t] = CAS then
8:       tsid = max{tsid, CIndex.lookup(ts.col_read_set[t])}
9:       tsid = max{tsid, RIndex.maxValue(t)}
10:    else if ts.class[t] = RAS then
11:      tsid = max{tsid, RIndex.lookup(ts.row_read_set[t])}
12:      tsid = max{tsid, CIndex.lookup(ts.col_read_set[t])}
13:    end if
14:  end for
15:  return tsid
16: end function

```

---

Table 5: The content of the Tables, Columns, and Rows Hash Indexes after executing the Table 1 transactions

<i>TIndex</i>	<i>CIndex</i>	<i>RIndex</i>
<i>R</i> → 14	<i>A3</i> → 13	( <i>A1</i> = 100) → 11
	<i>A4</i> → 13	( <i>A1</i> = 120) → 14
<i>S</i> → 15	<i>B2</i> → 12	
	<i>B4</i> → 15	

how the indexes can be used for finding the TSID of the last transaction that modified any of the data items accessed by  $T_r$ . For each table  $t$  to be accessed by  $T_r$ , we utilize the affecting classes and read sets for guiding our algorithm, assuming  $t$  has been modified before (lines#3-4). If  $T_r$  is a “TAS” for table  $t$  (i.e., it will access all columns of  $t$ ), then a single lookup on the *TIndex* is enough to find the latest TSID (lines#5-6). If  $T_r$  is a “CAS” for  $t$  (i.e., it will access some specific columns of  $t$ ), then we need to (i) lookup the *CIndex* to find the latest TSID among all columns in  $T_r$ ’s column read set and (ii) search the *RIndex* to find the largest TSID from all rows affecting  $t$  (lines#7-9). We search for the rows as well since any row modification can potentially modify any column. Finally, if  $T_r$  is a “RAS” for  $t$  (i.e., it will access some specific rows of  $t$ ), then we also lookup both *RIndex* and *CIndex* (lines#10-12). Overall, we return the largest TSID found from all lookups across all tables to ensure consistency. Any Extension DB that has replicated at least that TSID can be used for executing  $T_r$ .

Table 3 lists some example read transactions along with their consistent TSID based

on the indexes' content in Table 5. Consider transaction  $R_3$  that accesses columns  $B2$ ,  $B3$ , and  $B5$  of table  $S$ . Based on Table 5, 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.

## 7.2 Statement-level Load Balancing

Master-slave replication dictates that all transactions that modify the database, including multi-statement ones, must be executed on the master first. However, multi-statement write transactions may contain several read SQL statements, all of which are now executed on the Primary DB. Some of these reads could potentially be executed on Extension DBs without violating atomicity or consistency constraints and, hence, increase the scalability of the entire system.

The premise is that a read statement within a multi-statement write transaction  $T_m$  that is independent of its preceding write statements in  $T_m$  can be safely executed on a consistent Extension DB. This premise does not hold for serializable execution, but does hold for Snapshot Isolation, which is the default consistency level of Hihooi (see Section 7.3), because the read still sees a consistent snapshot of the database. Algorithms 6.1 and 7.2 can be used to efficiently check independence and find a consistent Extension DB, respectively. In particular, when the write statements of  $T_m$  are executed on the Primary DB, a running state is kept by the Transaction Manager (similar to the running state kept by the Extractors described in Section 6.3). When a read statement arrives in  $T_m$ , Hihooi checks if it is independent from the running state (recall Algorithm 6.1). If so, Algorithm 7.2 is used as in Section 7.1 to find the latest consistent  $TSID$ , and thus, the available Extension DBs for execution. When the read is not independent from the previous writes, or no consistent replica is found, it is executed on the Primary DB.

## 7.3 Consistency Levels

Most database engines (e.g., PostgreSQL, Oracle, DB2) use *snapshot isolation (SI)* for enforcing consistency [26]. 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 [60]. 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)** [60].

We follow concepts introduced in [16, 60, 37, 8] in order to formalize the notion of GSSI in replicated systems and develop a direct proof of its support by Hihooi. According to SI, as introduced in [16], the system assigns a transaction  $T$  a start timestamp  $s(T)$  at the beginning of its execution, before performing any read or write operations.  $T$  will always read data from a snapshot of the (committed) data as of  $s(T)$ . In particular, writes performed by any transaction  $T'$  that commits before  $s(T)$  will be visible to  $T$ . On the other hand, writes performed by any transaction  $T'$  that commits after  $s(T)$  will not be visible to  $T$ . SI also requires that each transaction  $T$  be able to see its own writes, even though the writes occurred after  $s(T)$ . After finishing its operations,  $T$  is assigned a commit timestamp,  $c(T)$ , such that  $c(T)$  is more recent than any start or commit timestamp assigned to any transaction.  $T$  commits only if all other transactions  $T'$  that committed during the lifespan of  $T$  (i.e.,  $s(T) < c(T') < c(T)$ ) did not modify any data that  $T$  has also written. Otherwise,  $T$  is aborted so as to prevent lost updates. Note that two transactions  $T_1$  and  $T_2$  are called *concurrent* if their lifespan intervals  $[s(T_1), c(T_1)]$  and  $[s(T_2), c(T_2)]$  overlap.

According to the original definition of SI, the system can choose  $s(T)$  to be any time less than or equal to the actual start time of  $T$ . Hence,  $T$  can see any snapshot earlier than its start timestamp and not necessarily the latest one. This relaxed version of SI is called *Weak SI* in [37]. With *Strong SI*, a transaction  $T_2$  that starts after a committed transaction  $T_1$  is guaranteed to see a committed database state that includes the effects of  $T_1$ . In other words,  $T_2$  will see the latest snapshot of the database state. Most current database systems (including PostgreSQL) and research prototypes [110, 73, 77] offer Strong SI. Finally, the qualifier '*global*' indicates that the definition of Strong SI applies to the distributed system as a whole and not to the individual database replicas.

Summarizing, a transaction history in a replicated database system satisfies Global Strong SI if its committed transactions satisfy the following two conditions:

1. Read operations in any transaction  $T$  see the database in the state after the last commit before  $s(T)$ . Read operations in  $T$  also see the data values that were last written by  $T$  itself;
2. Concurrent transactions do not modify the same data objects in the database.

**Theorem 1.** *If each underlying database system in the replicas guarantees Strong SI, the Hihooi guarantees Global Strong SI.*

*Proof.* Given a set of transactions to be executed with Hihooi, we need to show that their transaction history will satisfy the two conditions of Global Strong SI noted above.

**Condition 1:** Suppose  $T$  is a read transaction arriving for execution in Hihooi. Algorithm 7.2 will find the TSID of the last write transaction that modified any of the data to be accessed by  $T$  (recall Section 7.1). Then,  $T$  will be routed to any Extension DB that replicated at least that TSID, guaranteeing that  $T$  will see the latest relevant state. When no such Extension DB is found,  $T$  is routed to the Primary DB, which is always up to date and offers Strong SI by itself. If  $T$  is a write transaction, then it will be executed on the Primary DB. Since all write transactions always execute on the Primary DB and the Primary DB guarantees Strong SI locally, any read operations in  $T$  will see the latest database state. With statement-level load balancing (recall Section 7.2), read statements in  $T$  that are independent of their preceding write statements in  $T$ , can be routed to Extension DBs. Since the routing algorithm is the same as the one used for read-only transactions, these read statements will see the latest relevant state as explained above. Read statements in  $T$  that access data written by previous write statements in  $T$  are sent to the Primary DB and, hence, will see the data values that were last written by  $T$ .

**Condition 2:** Since all write transactions are always executed on the Primary DB, which offers Strong SI, no concurrent transactions can modify the same data and commit successfully on the Primary DB. On the Extension DBs, transactions that modify the same data are never run concurrently per Algorithm 6.2 (recall Section 6.3). Only independent

write transactions are ever executed in parallel on the Extension DBs, guaranteeing that concurrent transactions do not modify the same data in the database. □

By controlling the replication and routing mechanisms, Hihooi can offer three additional consistency levels at the granularity of a database session:

1. **Weak SI:** Write transactions are asynchronously executed on the Extension DBs and read transactions are sent to any Extension DB regardless of their consistency.
2. **Replicated SI with Primary Copy (RSI-PC):** 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). RSI-PC, another form of GSSI, is implemented by the middleware Ganymed [77].
3. **One-copy Serializability (ISR):** Write transactions are synchronously executed on all Extension DBs and read transactions are sent to any Extension DB. *ISR is supported by Hihooi as long as the master database uses serializable transaction isolation.* The entire write workload goes to the primary first so the generated schedule is serializable. Then, it is replicated in the Extension databases and it is applied in the primary's commit order; hence, the schedule there is serializable (the same serial order applies). Finally, the reads executed at the Extensions are routed there only if a replica is up to date. The order of execution of the reads does not matter as they don't conflict with any other running transaction or with each other, and hence, any serial order works. ISR is the default consistency level of middleware C-JDBC [27].

## 8 Scalability Management

Starting and shutdown the system, adding or removing Extension DBs and performing backups are important management operations. During starting and shutdown operations, the system is moved from one state to another and requires extra consistency checks to ensure that the consistency between Primary DB and Hihooi is retained. Furthermore, adding/removing Extension DBs and performing backups are important management operations for ensuring Hihooi's fault recovery and proper scalability. This section explains these operations and discusses some enabling (future) work on automated backup and elasticity management. Finally, it presents common scenarios in order to improve how the system provides durability against various failures.

### 8.1 Hihooi Startup and Planned Shutdown

**Startup:** During the Hihooi startup, the state of the system could be either using a new system checkpoint or can be a normal startup after a planned shutdown. In the former state, the system is already synchronized with Primary DB (due to the new SEED DB restore from Primary DB), and Extension DBs could be created on demand from the new created SEED DB. As a result, in this state there are no consistency worries and no consistency check is required. In the latter state, the system could restart to its previous state, i.e., the state before the point where it ended. Initially the system checks that Transaction Manager and Primary DB agreed on the Last Commit TXID. The Transaction Manager gets its own latest TXID by reading the tslog file entry (TSID:TXID). However, the Primary DB's latest TXID is selected from the database, during the check. If both TXIDs are equal, the system can start, otherwise, the system automatically starts recovering using the txlogs file, i.e., reading from txlogs the entries TXID,sql\_text,timestamp and loads it into Transactions State for those TXID completed after the TXID located in tslog. After the Transaction Manager starts up, the system can reuse all or a part of the Extension DBs used in the previous run (note that Extension DBs are not destroyed after a plan shutdown). TXID requirement is not valid for Extension DBs, because they are not necessary to implement the last TSID in order to rejoin the system. The only requirement is

to belong in the same version of SEED DB. The consistency between the Extension DBs and Transaction Manager is determined by the last TSID replicated in Extension DBs and the version of SEED DB (i.e., all used Extension DBs should have the same SEED DB version). When the Hihooi shutdown, the Transaction Manager and each Extension DBs save in disk (i.e., into tslog file) the latest TSID as a reference point for the next restart. Also, an Extension DB is not prevented from entering the system with a smaller TSID number, because, the Transaction Manager ensures that have all missing Transaction Sets are available, so all the Extension DBs could be synchronized with the latest TSID.

**Planned Shutdown:** Hihooi and Primary DB are two separated entities and as a result, their shutdown procedures are not correlated. In the case of Hihooi shutdown, the Primary DB remains open, and it is obvious that the Primary DB is vulnerable for external interference. In other words, Hihooi needs to confirm that the Primary DB's state remains the same and after Hihooi's restart. As a result, both systems' shutdown are necessarily binding. On Hihooi startup, the reconciliation phase guarantees that the Hihooi's latest commit TXID is equal with Primary DB's last commit TXID. It is also important to mention how to close the system so as not to affect the system's consistency. (1) Shutdown Normal: This is the default shutdown procedure. In this mode, the Transaction Manager does not accept any new connection and waits for all current transactions to become Replicated and Archived. Thereupon, the system shutdowns all Extension DBs, and the Primary DB. (2) Shutdown immediate: In this mode, the Transaction Manager does not accept any new connection and waits for all current transactions to complete (either commit or rollback). Thereupon, the system shutdowns all Extension DBs, and Primary DB. All transactions that are not replicated to Extension DBs are recovered during the next startup.

## 8.2 System Initialization, Backups and Fault Recovery

The process of initializing the first system's replica is called *system checkpoint* and includes the following steps:

- 1) Create a Primary DB consistent Backup called *Seed DB*.
- 2) Create a number of Extension DBs by cloning the Seed DB.
- 3) Start Hihooi components (i.e., Transaction Manager, Transaction Buffer and Extension DBs) by resetting TSID to zero.

The Seed DB is created using the Primary DB's vendor utility. In order to keep the content of the database intact, the Primary DB does not support any transactions during the backup. Every new Seed DB is sequential number, starting from one, associated with the Primary DB's latest system change number (SCN) e.g., the last commit timestamp or last commit transaction id (TXID). The sequential number helps the system to correlate the Seed DB with the Primary DB. The Extension DBs are created by cloning the Seed DB.

During and after the Seed DB creation, write transactions are modifying the Primary DB and are recorded into the Transactions Buffer. As a result, all new Extension DBs that will be cloned from SEED DB will have a huge number of Transactions Sets to implement, because SEED DB is starting from TSID equal 0 and the rest system is more forwarded. If the number of TSIDs is too large, in relation with SEED DB's TSID, the new Extension DB's synchronization will take more time to complete. It is also possible that part of TSIDs that are needed for synchronization will not exist in Transaction Buffer, as a result, it takes another time from Archiver process to load it again in Transaction Buffer from the archives. The time that takes for an Extension DB to reach the latest system's TSID is called Extension DB Recovery Time (`EXTDB_TIME_TO_RECOVERY`). The `EXTDB_TIME_TO_RECOVERY` is measurable because, Hihooi includes in the Transaction Set the Transaction's elapsed time in Primary DB. In order to reduce the TSID's gap between the current state of the system and SEED DB's state and make Extension DB synchronization faster, we suggest one, or a combination of the following solutions:

1. Automatic SEED DB replacement in case that `EXTDB_TIME_TO_RECOVERY` parameters exceeded some predefined value, e.g., 30 minutes. This solution requires a standby Extension DB in RECOVERY mode which will replace the SEED



DB when the `EXTDB_TIME_TO_RECOVERY` is exceeded. This option helps the standby Extension DB to quickly catch up to the last state, however, it overprovisions system's resources. However, this option is very helpful, when the system needs to scale up very quickly.

2. SEED DB replacement after a normal shutdown with a FULL synchronized Extension DB. Furthermore, if the system has a shutdown window, the system administrator can replace SEED DB with a FULL synchronized Extension DB.
3. SEED DB replacement with an Active Extension DB. This system's feature is possible, but requires removing a production machine in order to replace the SEED DB. However, this action will affect system's performance if it is done at a time when the system needs all the production machines. This replacement can only take place while the system no longer needs some machines.
4. Shutdown System without destroying the Extension DB members. For example, if the system has a predefined number of Extension DB members (e.g., minimum and maximum number of Extension DB member), then the already instantiated Extensions DBs will not be destroyed on system's shutdown. This option helps the system to quickly restore to the previous state, however, it overprovisions system resources.
5. Frequently system checkpoint. If the Primary DB is small, and the system checkpoint does not take long time to complete, as well as , frequent system shutdowns do not influence the production flow, frequently system checkpoints is a good solution.

Through an example, we will highlight the resilience of the system to the following scenarios:

1. Transaction Manager Failure.
2. Transactions Buffer Failure.
3. Primary DB Failure.

System Variables		TXState			
Name	Value	TXID	TSID	Time	Status
PRM DB Last Committed TXID	711	714			Running
Hihooi Last TSID	2	713			Running
Extension 1 Last Applied TSID	1	712			Running
		711		T2	Committed
		710	2	T1	Committed Archived
		709	1	T0	Replicated
<b>Transaction Manager tslog</b>		<b>Extension 1 tslog</b>			
710 t1 2		709 1.1			
<b>Transaction Manager txlogs</b>		<b>Transaction Manager Archived Files</b>			
711 SQL_TEXT_3		1.2.arc			
711 COMMIT t2		1.1.arc			
710 SQL_TEXT_1					
710 COMMIT t1					
709 SQL_TEXT_2					
709 COMMIT t0					

Figure 10: Hihooi state snapshot at time t2

#### 4. Extension DB Failure.

Figure 10 illustrates a system's snapshot on important variables, memory structures, system files and directories on time t2. Based on the system's snapshot the last committed TXID is the TXID 711 which committed on time t2 and the last Hihooi TSID is TSID 2. The Transaction with TXID 710 is committed and archived by T-ARC but not yet replicated, the Transaction with TXID 709 is replicated and applied in Extension 1. The Transactions 712 to 714 are still running. In our scenario, the Transaction Manager's tslogs at time t2 contains only the last replicated TSID 2, because the system did not yet calculate the already committed transaction 711. However, Transaction Manager's txlogs contains all the Transaction logs.

**Transaction Manager Failure:** If the Transaction Manager's failed on time t2 e.g., instance failure, on the next Transaction Manager's Startup, an automatic recovery will take place because, the TXID in the Transaction Manager's tslog is different from the Primary DB last commit TXID. The Transaction Manager will reload the Transactions State (by reading txlogs), all the transactions and their status which are greater than t1, in our case, the Transaction 711, with status committed. Also, the Transaction Manager will invoke Archiver to reload in Transaction Buffer the Transaction sets 1.2. i.e., the difference between its own TSID and Extension 1 TSID (This check is completed on

every Transaction Manager startup or its triggered when new Extension DB added, in order to supply the system with missing Transaction sets, if any). The rest transactions i.e., 712 to 714 will be aborted by Primary DB due to the Transaction Manager's failure. However, when the recovery completes, the system will get its normal way.

**Transactions Buffer Failure:** If the Transactions Buffer fails, the Extension DBs will stop receiving any updates but will still be able to serve any read transactions that are consistent with their current database state. Upon recovery of the Transactions Buffer, all collected TStates will be pushed from the Transaction Manager to the Transactions Buffers and from there they will be applied on the Extension DBs in parallel.

**Primary DB Failure:** In case of a Primary DB failure, the Transaction Manager in order to protect the consistency of the system will signal a shutdown normal operation (recall, in this mode, the Transaction Manager does not accept any new connection and waits all current transactions to become Replicated and Archived. Thereupon, the system shutdown all Extension DB). In our example, the system's next steps during the shutdown normal operation will be (i) the Transaction TXID 710 will be replicated, (ii) Transaction TXID 711 will be archived, (iii) a new TSID number 3 will be created, and (iv) the 1.3 Transaction set will be replicated to Extension DB 1. On the next system's startup the system will start normally without any recovery.

### **8.3 Adding and Removing Extension DBs**

The addition of a new Extension DB involves two steps: (i) the replication of the Seed DB on the new node, and (ii) the parallel re-execution of all transactions located on the Archiver Buffer using the procedure described in Section 6.3. Afterwards, even though the new Extension DB might not be fully consistent with the Primary DB, it will register with the Transaction Manager and start serving consistent read requests, while applying the write transactions from the Transactions Buffer. Hence, the addition of a new replica in Hihooi does not require a global synchronization barrier or the use of resources from other active replicas [26].

Extension DBs may be removed from the system for a variety of reasons such as maintenance operations, insufficient workload to justify their presence, and failures. Since Extension DBs only serve read transactions to the clients, no complicated failure mechanisms are needed from the client's perspective. The Transaction Manager is either notified or detects the removal of an Extension DB and simply re-routes the read transactions to other consistent Extension DBs. During the application of the write transactions from the Transactions Buffer, the Extractors log all completed transactions. When the node is re-added to the system, the write transactions are replayed from that point forward.

The addition or removing of an Extension DB from the system, requires a redistribution of the existing connections to the instantiated Extension DBs. Without this feature the system will not be able to utilize equally all the instantiated Extension DBs. The implication of the absence of the workload's rebalancing will result in poor workload scalability performance. As a result, when the number of active Extension DBs changes in the system, the Transaction Manager's routing algorithm *safely* (i.e., priority is given to maintaining strong consistency) redistributes all the current connections to the available Extension DBs.

## **8.4 Towards Replica Self-Management**

The time required to start an Extension DB depends on the time needed to replicate the Seed DB (if the new Extension DB starts on a new node) plus the re-execution time of the write transactions on the Archiver Buffer. The former can be easily calculated since the process just entails bulk I/O transfers of known sizes. The latter can also be computed because the execution time of each write transaction is recorded into the TState. Hence, Hihooi can accurately model and estimate the replica synchronization time. This model can guide the decision on how frequently to create new backups in order to provide bounded guarantees on the time needed to deploy a new Extension DB.

The ability to add and remove replicas without service interruption in addition to accurately modeling their cost are key steps towards autonomic middleware-based replicated databases. Recent work on database workload monitoring and characterization

(e.g., [36, 40]) could guide the development of elasticity policies that automatically decide when to add or remove nodes based on the actual workloads. Another interesting future direction would be integrating Hihooi with the cloud, which would extend the type or resources available for hosting the replicas. Finally, cloud technologies such as Virtual Machine migration or cloning could be used for creating the backups and launching new Extension DB nodes.

## 9 Experimental Evaluation

The purpose of our evaluation is (1) to evaluate the system’s performance and scalability under varying workload types and consistency levels, (2) to study the effects of our fine-grained statement replication and routing algorithms, and (3) to evaluate the key management and fault tolerance features of Hihooi. All experiments were run on a 13-node cluster running CentOS Linux 7.2 with 1 Primary DB, 1 Seed DB, 8 Extension DBs, and 3 client nodes (with up to 16 clients each). The Transaction Manager is running on the Primary DB node, its backup and Archiver on the Seed DB node, and the Transactions Buffer on an active Extension DB node. The primary node has an 8-core, 3.2GHz CPU, 64GB RAM, and 2.1TB HDD storage. The rest nodes have an 8-core, 2.4GHz CPU, 24GB RAM, and 1.5TB HDD storage.

For our evaluation, we used three well-known benchmarks, each employing a different kind of workload: (i) **TPC-C** [101], the industry standard for OLTP workloads, containing complex and write-intensive transactions; (ii) **YCSB** (Yahoo Cloud Serving Benchmark) [33], a collection of web-based micro-benchmarks that represent data management applications whose workload is simple but requires high scalability; and (iii) **CHB** (CH-benCHmark) [31], a workload combining OLTP from TPC-C and OLAP from TPC-H [102].

The TPC-C database (also used by CHB) was populated with 500 warehouses for a total size of 50GB. For the YCSB database, we used a scalefactor of 50000, resulting in 56GB of data. The databases were fully replicated to the Extension DBs. We used PostgreSQL version 9.5.3 in all nodes. The results presented, unless noted otherwise, are from 10 minute trials, preceded by 2 minutes of warm up. OLTP-Bench [39] was used to populate and run the tests for all benchmarks. The transactions load was injected using 6 clients per 1 Extension DB that continuously issued transactions.

### 9.1 OLTP Workload Scalability

This section studies the effectiveness and efficiency of Hihooi in scaling an OLTP workload by measuring its throughput and latency as we increase the number of Extension

Table 6: Composition of TPC-C workload mixes

Transaction	Read-Only	Read-Heavy	Balanced	Write-Heavy
New-Order			2%	7%
Payment		3%	3%	5%
Order-Status	50%	50%	85%	73%
Stock-Level	50%	47%	10%	15%

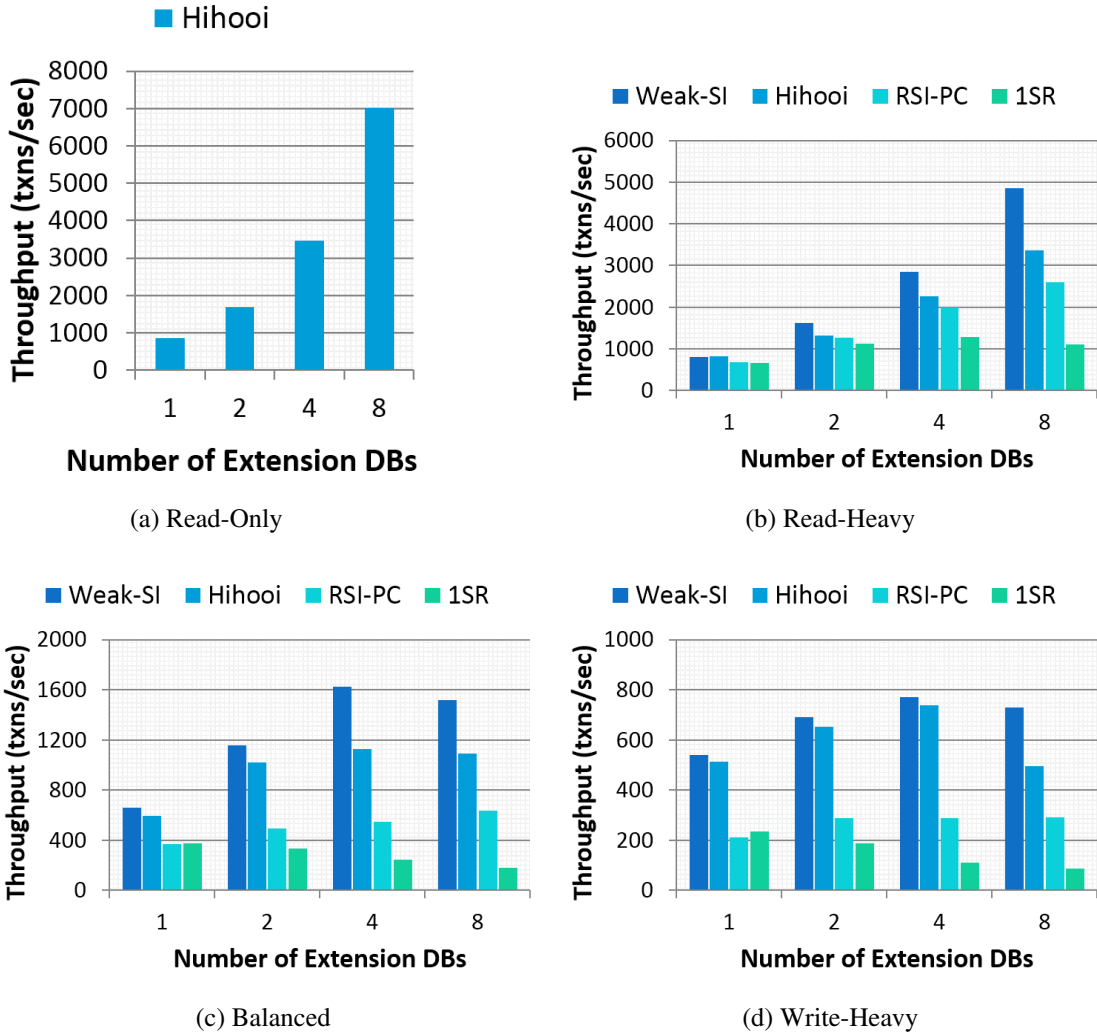


Figure 11: OLTP workload scalability for TPC-C for different workload mixes and consistency levels

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 7.3). 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.

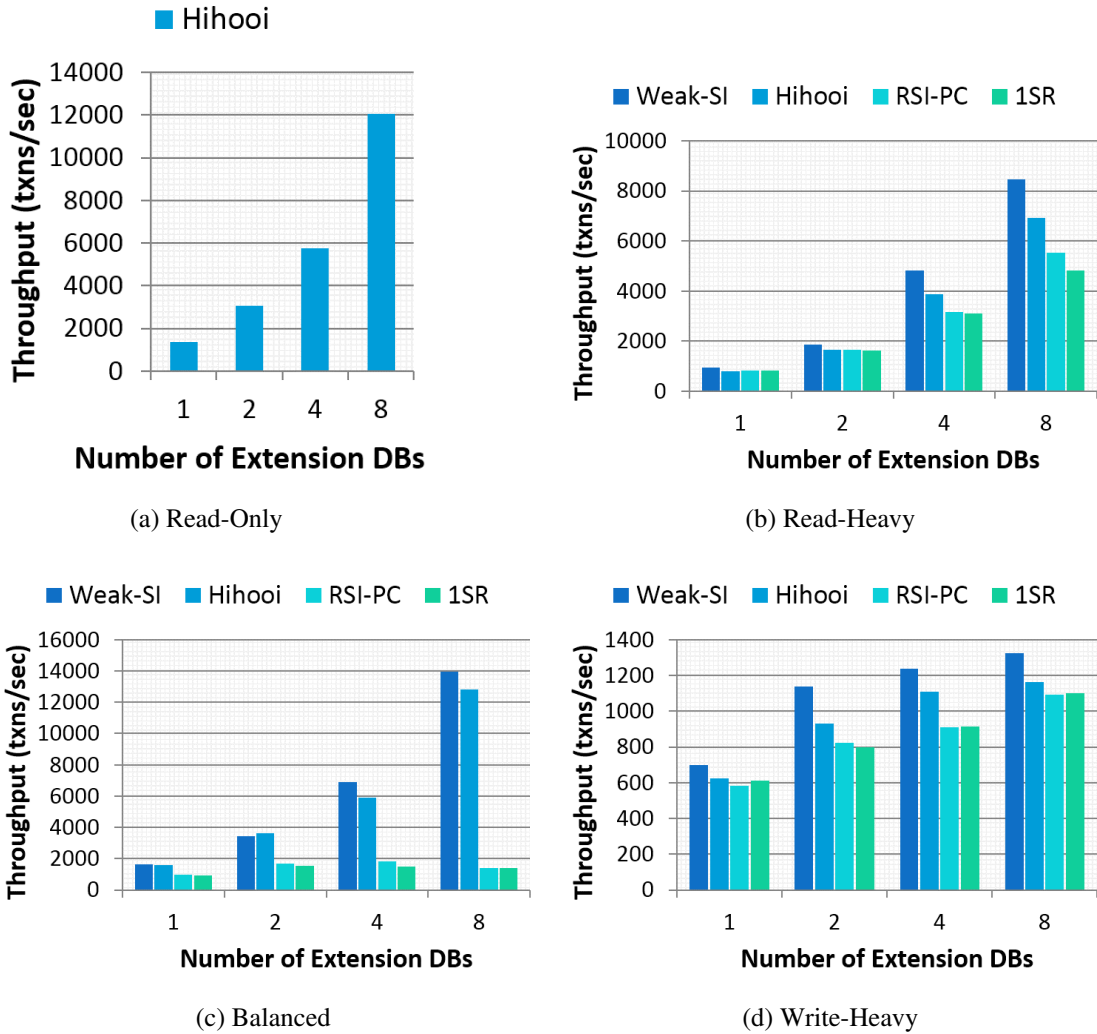


Figure 12: OLTP workload scalability for YCSB for different workload mixes and consistency levels

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 statements, respectively, and were generated via mixing the TPC-C transactions as shown in Table 6. Figure 11 shows the throughput rates in committed transactions per second for our workload mixes and consistency levels. 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



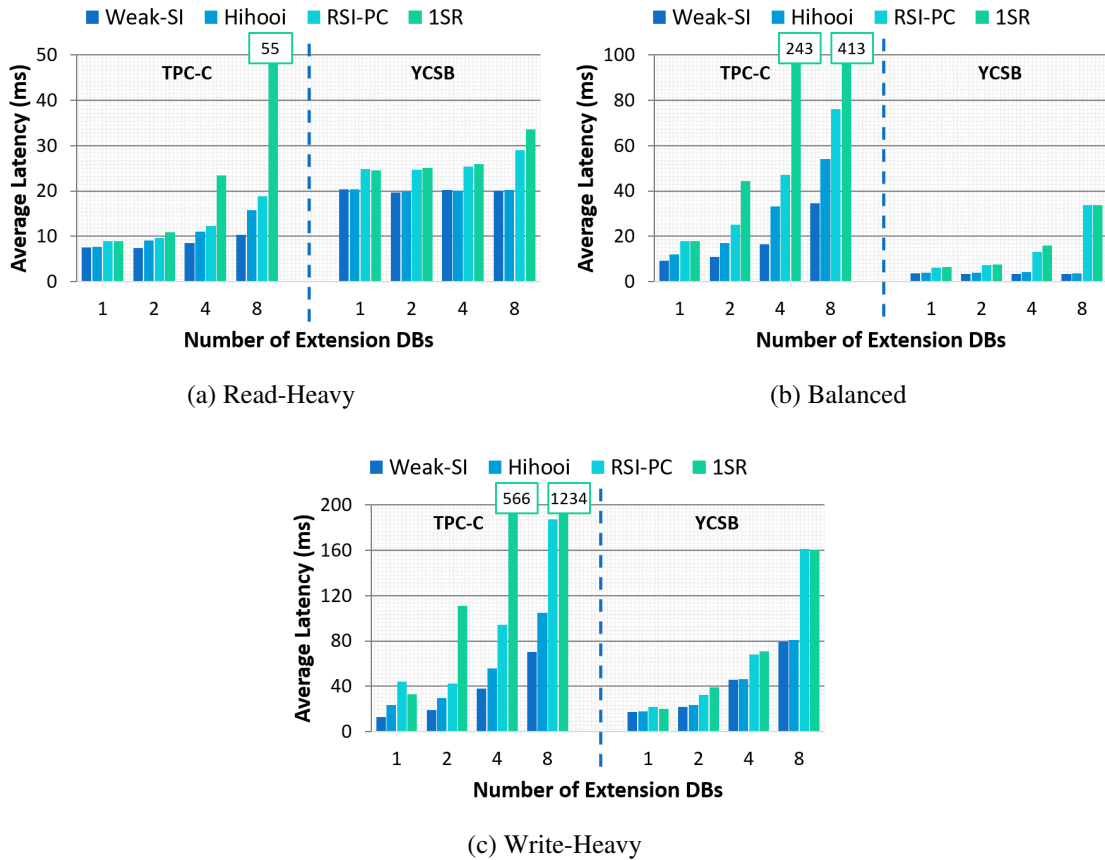


Figure 13: Average latency for TPC-C and YCSB for different workload mixes and consistency levels

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.

Figure 12 shows the throughput rates for four workload mixes and our consistency levels for YCSB. All YCSB workloads follow a Zipfian distribution ( $\theta=1$ ) and contain unmodified queries. Similar to TPC-C, the Read-Only YCSB workload exhibits almost perfect linear scalability. The Read-Heavy workload consists of 5% inserts and 95% range scan queries, per [33], while the Balanced workload consists of 85% single-row reads and 15% inserts. Both *Weak-SI* and *Hihooi* are still able to achieve near linear scalability, while *RSI-PC* and *1SR* do not scale at all for the Balanced workload. Based on our observations, inserts in YCSB are 2-3x faster than reads. In *Ganymed*, this results in delays in the execution of reads, which need to wait for all fast inserts to propagate to at least one replica. The use of row R/W sets by *Hihooi*'s routing algorithms excels

in this test as it enables the system to route a read transaction  $T_r$  to a replica that might not be fully consistent with the Primary DB but is consistent for  $T_r$ . Finally, the Write-Heavy workload consists of 50% reads and 50% updates, per [33]. As this is a more demanding workload, both throughput and scalability suffer. Nevertheless, *Hihooi still performs considerably better (up to 1.42x) compared to RSI-PC and ISR for the same reasons.*

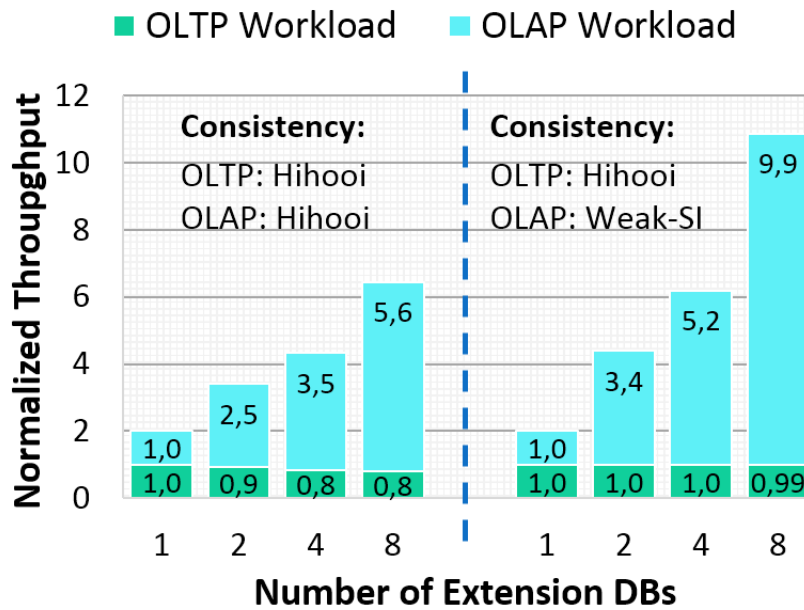


Figure 14: Mixed OLTP-OLAP workload scalability for CHB

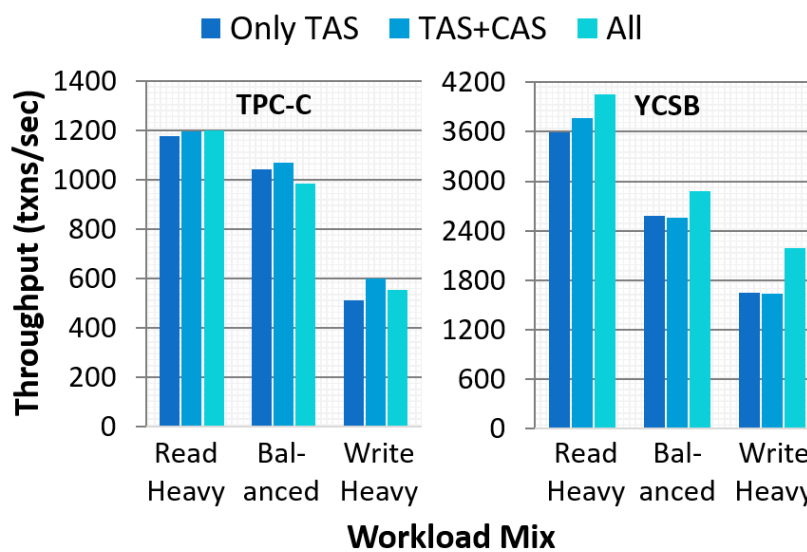


Figure 15: Effect of using the TAS, CAS, and RAS classes

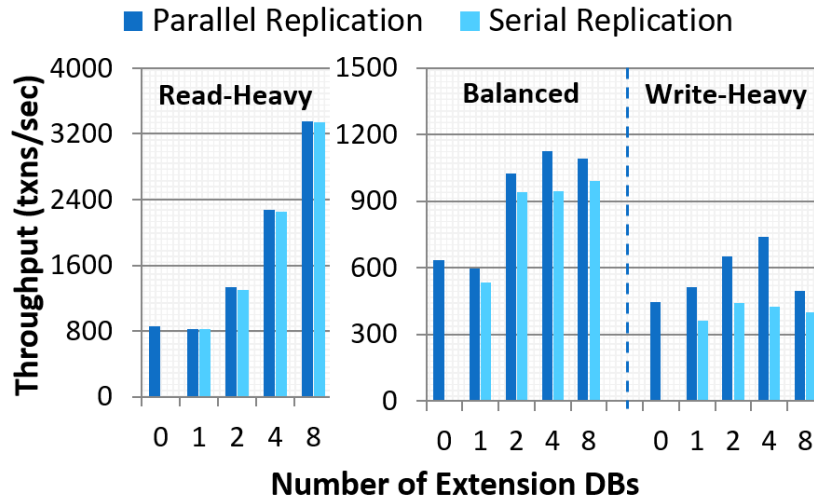


Figure 16: Effect of Hihooi's parallel replication algorithm on TPC-C

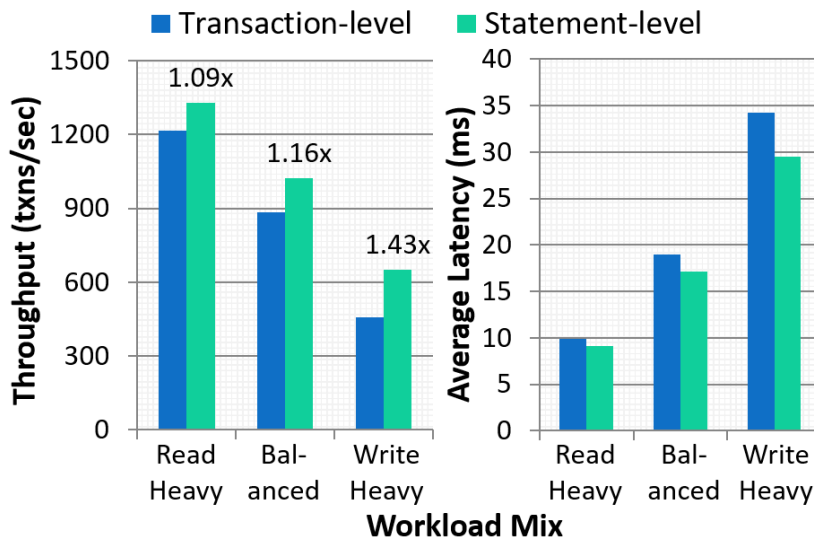


Figure 17: Effect of statement-level load balancing on TPC-C

Figure 13 shows the average latency of transactions across different workloads mixes for TPC-C and YCSB. For the Read-Heavy workloads, there is very little to no increase in latencies as the number of replicas increases due to the efficient load balancing of read queries to the replicas. However, as more writes are introduced in the Balanced and Write-Heavy workloads, adding more replicas increases the average latencies (even for Weak-SI) as a bigger percentage of the workload is sent to the Primary DB. Focusing on Hihooi, we observe only a small increase in latency as the number of replicas increase, indicating the low overhead added due to replication. Conversely, both RSI-PC and 1SR cause increasingly larger latencies for all workloads due to waiting reads (as explained

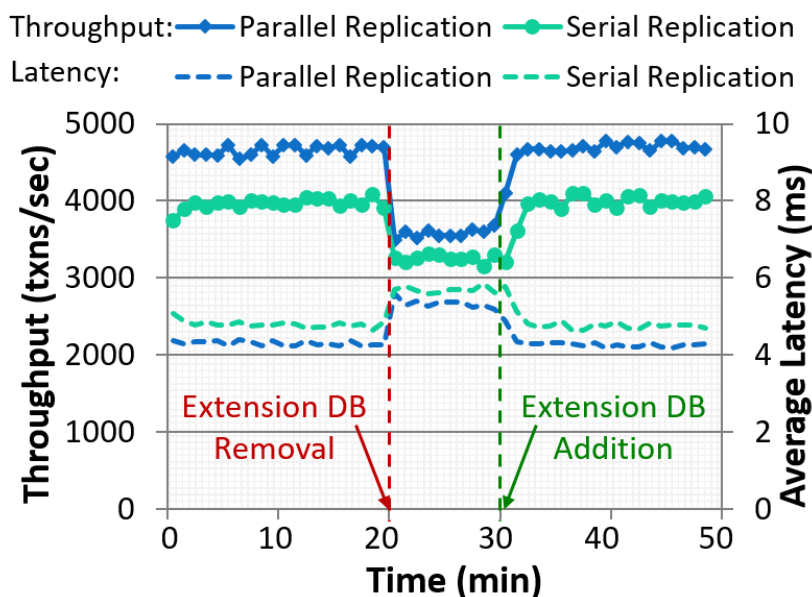


Figure 18: YCSB throughput after removing & adding 1 Extension DB

above). Once again, *Hihooi* is able to offer latencies comparable to *Weak-SI* in almost all cases due to its fine-grained transaction routing capabilities.

## 9.2 OLTP-OLAP Workload Scalability

The execution of OLAP queries on transactional databases has long been a motivating scenario for database replication [26, 60]. In this section, we evaluate the OLAP workload scalability provided by *Hihooi*, while studying its effects on an OLTP workload. For these tests, an OLTP client node executes the CHB transactional workload, while two OLAP client nodes submit the CHB analytical queries, all using the default *Hihooi* consistency level (GSSI). The general trend, as shown in Figure 14, is that the OLAP workload scales sub-linearly, while the OLTP one exhibits a small negative impact that worsens as the number of replicas increases (7-20%) because more OLAP queries are forced to execute on the Primary DB (due to read-write conflicts). However, since OLAP workloads do not typically require strong consistency, we repeated the experiment using *Weak-SI* for the OLAP workload and GSSI for the OLTP one. The new results (see Figure 14) reveal *linear scalability for the OLAP workload with almost no overhead for the OLTP one*, and highlight the great benefits offered by *Hihooi* in this setting.

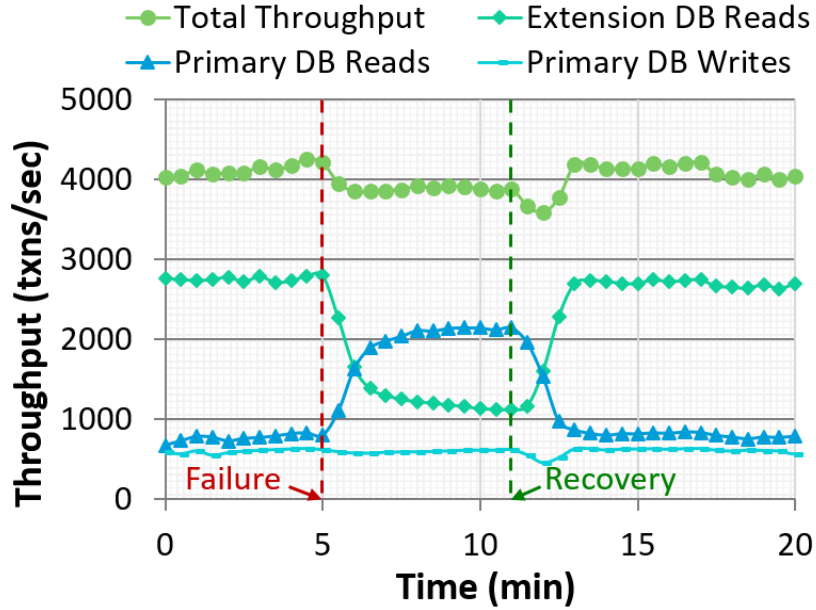


Figure 19: YCSB throughput during Transactions Buffer failure

Table 7: Percentage (%) of TAS, CAS, and RAS statements

Benchmark	Workload	Affecting Class		
		TAS	CAS	RAS
TPC-C	Read-Only	0	10	90
	Read-Heavy	4	10	86
	Balanced	23	14	63
	Write-Heavy	40	12	48
YCSB	Read-Only	0	0	100
	Read-Heavy	95	0	5
	Balanced	15	0	85
	Write-Heavy	50	0	50
CHB	OLTP	56	13	31
	OLAP	82	18	0

### 9.3 Effect of Affecting Classes

Query statements can modify or access data at different levels of granularity, namely at the table, column, or row level, captured by our definitions of TAS, CAS, and RAS classes, respectively (recall Section 6.1). Table 7 lists the percentage of TAS, CAS, and RAS statements for each workload mix of each benchmark. All TPC-C workloads contains a mix of all three types of statements, while the percentage of TAS increases as the workload becomes more write-heavy. YCSB workloads, on the other hand, contain a significant fraction of RAS and no CAS statements.

This section studies the effect of letting Hihooi use these increasing levels of granularity by configuring it to use: (i) only TAS; (ii) TAS and CAS; and (iii) all classes. We

executed both the TPC-C and YCSB workloads with our different mixes on Hihooi using two Extension DBs. The results presented in Figure 15 reveal that *the benefits from utilizing the CAS and RAS classes depend both on the read-write mix and the workload itself*. In particular, the benefits for read-heavy workloads are relatively small because the replicas are almost always consistent with the Primary DB and the reads are typically load-balanced regardless of their affecting class. As the write portion of a workload increases, there are more opportunities for Hihooi to route read statements that access some columns or rows of a table, even though some other columns or rows of that same table have been modified. This is more evident with YCSB, whose Balanced and Write-Heavy workloads contain a significant fraction of RAS statements. Hence, when Hihooi is able to exploit RAS, the overall throughput is increased up to 32% in our experiments. TPC-C on the other hand, uses more complex transactions that effect tables both at the column or row level, leading to small benefits from using CAS or RAS (less than 22%). The effects on average latency follow the same trends as the effects on throughput shown in Figure 15 and are not shown due to space constraints. Finally, the memory overhead from the hash indexes is very low as the maximum one measured in all experiments was less than 300 KB.

#### **9.4 Parallel Replication Algorithm**

This section delves into the performance implications of the parallel replication algorithm (recall Section 6.3) compared to the common approach that executes the write transactions serially on the replicas. The two approaches have no to little impact on the throughput of the Read-Only & Read-Heavy TPC-C workloads (see Figure 16) 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. The actual degree of parallelism ( $dp$ ) when applying the writes depends on the submission order of the writes as well as the portion of the data they apply to. As an indicative example, the average  $dp$  for the Balanced workload with 4 Extension DBs was 4. 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 generated concurrently by the 48 clients overload the Primary DB. At that point, even the Weak-SI case experienced a performance drop (see Figure 11(d)).

The 0-Extension DBs setting in Figure 16 corresponds to processing the workload on a single node without replication. The difference between having 0 and 1 Extension DBs reveals the overhead incurred by Hihooi from intersecting all transactions, which was typically low (<4%) and no more than 9% across all experiments (not shown due to space constraints). It is interesting to note that for heavy-write workloads, Hihooi is very effective in separating the execution of writes and reads on the Primary and Extension DBs, respectively, leading to an aggregated higher throughput.

## 9.5 Statement-level Load Balancing

One of the most novel aspects of Hihooi is its ability to route individual read statements to consistent replicas, even within multi-statement write transactions. This section evaluates the effect of statement-level versus (the typical) transaction-level load balancing, which always routes all statements from a write transaction to the Primary DB. Figure 17 shows the throughput and average latency for our 3 TPC-C workloads executed on Hihooi running with two Extension DBs using either transaction- or statement-level load balancing. *As the percent of writes increases, so does the benefit of statement-level load balancing*, leading up to 1.43x better throughput and 14% lower latency compared to transaction-level load balancing. These benefits are attributed to the extra read statements that are diverted to the Extension DBs. Specifically, in the Write-Heavy workload, the transaction-level algorithm routes 29% of the reads to the Primary DB either because there are no consistent Extension DBs or the reads are part of multi-statement write transactions. On the contrary, the statement-level algorithm routes only 15% of reads to the Primary DB, while the remaining are load balanced to the Extension DBs. Overall, with more multi-statement write transactions, Hihooi has more opportunities to divert the in-

cluded reads to Extension DBs, increasing parallelism and, therefore, throughput.

## 9.6 Adding and Removing Extension DBs

Next, we explore the scenario of adding and removing an Extension DB at run time. We started running the Balanced YCSB workload on Hihooi with 18 Clients and 3 Extension DBs. After 20 minutes, we removed 1 Extension DB to simulate a failure or planned maintenance operation. The read transactions executing on the Extension DB failed at that point but the Transaction Manager automatically rerouted them to other replicas. Hence, Hihooi continued serving the workload without any issues, albeit with a 24% lower throughput and higher average latency, as shown in Figure 18. After 10 minutes, we restored the Extension DB and observed the throughput rate return to its normal level quickly. *The Extension DB was able to serve its first read just 64 seconds after restoration* due to our fine-grained routing algorithm, while it was able to apply all changes it missed during the outage in 82 seconds. In total, it missed 321786 write transactions, while the memory size of the Transactions Buffer grew to only 384MB. We repeated the above procedure using the serial replication approach and observed a lower throughput and higher average latency during the entire experiment, while it took the Extension DB 121 seconds to catch up; highlighting once again the benefits of our parallel replication algorithm.

## 9.7 Transactions Buffer Failure and Recovery

In this section, we investigate the behavior of Hihooi during the failure and recovery of the Transactions Buffer. We started running the Balanced YCSB workload on Hihooi with 24 Clients and 4 Extension DBs. After 5 minutes, we induced a failure on the Transactions Buffer, which caused the Extension DBs to stop receiving any updates. However, Hihooi kept serving the incoming workload without any query failures. As the Extension DBs kept falling behind, the amount of read transactions executing on them decreased (since the YCSB workload is skewed to favor recent items), while many read transactions were routed towards the Primary DB, as shown in Figure 19. The write throughput was unaffected by the failure due to the asynchronous nature of the replication procedure. Overall,



the total throughput experienced a small slowdown of only 6.2%. After 6 minutes, we recovered the Transactions buffer. At that point, all Transaction States accumulated at the Transaction Manager (207220 in total, 247MB in size) were pushed on the Transactions Buffer and the Extension DBs started applying them in parallel. The overhead caused by the recovery process led to a small, 5.4% decrease in the overall throughput of the workload, which lasted for only 97 seconds until it returned back to its pre-failure level. These results show that Hihooi is able to gracefully handle a Transactions Buffer failure.

## 9.8 Comparison with PostgreSQL Replication

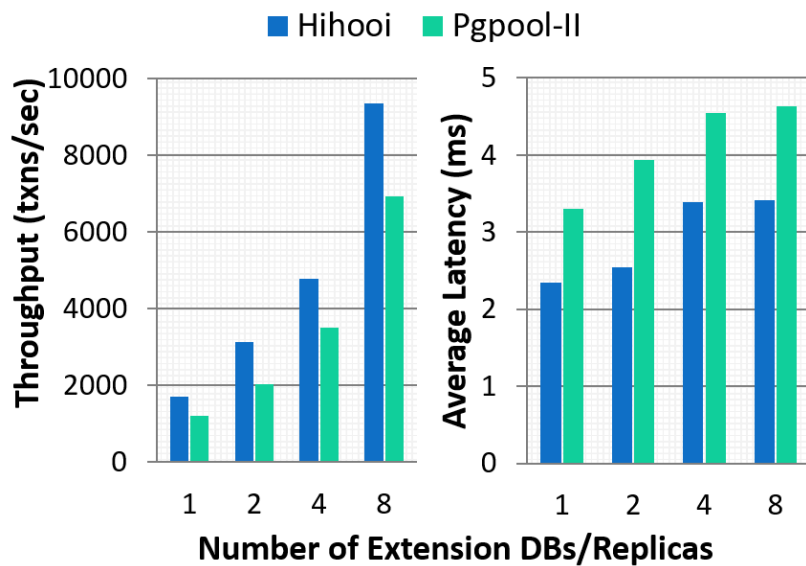


Figure 20: Pgpool-II Vs Hihooi for YCSB Balanced workload

PostgreSQL supports master-slave replication, where the master database server executes both read and write transactions and the slave (“hot standby”) replicas execute only read queries. PostgreSQL replicates database modifications via streaming WAL records from the master to the replicas, i.e., it employs *row-based replication*. This replication is asynchronous by default so the data on the standby is eventually consistent with the primary. On top of a PostgreSQL cluster, Pgpool-II [63] is used to provide connection pooling and load balancing of read queries to the replicas. We have setup Pgpool-II with PostgreSQL replication on our local cluster and compared its performance against Hihooi.

Figure 20 shows the throughput and average latency of the YCSB Balanced workload

when executed on Pgpool-II and Hihooi as the number of replicas increases. Our results show that both systems can scale throughput with more replicas in a similar fashion, while having a small negative impact on the average latency. Nonetheless, Hihooi is able to offer 35-55% higher throughput and 26-35% lower latency compared to Pgpool-II across all experiments. Hihooi's better performance is attributed to (i) its parallel replication algorithm (as PostgreSQL applies the WAL records serially), and (ii) its fine-grained routing algorithm (as Pgpool-II does not load-balance multi-statement write transactions).

## 10 Preliminary Investigation of Future Work

The ability to add and remove replicas without service interruption as well as rebalancing the workload automatically (recall Section 8) are necessary steps towards the creation of a truly autonomic middleware-based replicated database system. The key piece missing is the development of elasticity policies that automatically decide when to add or remove nodes based on the actual workloads.

**Definition 10.1.** *We define a workload  $W$  as a set of  $n$  SQL Statements that can be categorized using Hihooi's Affected Classes  $\ni$  (RAS,CAS and TAS) and executed as a set of Transaction  $T_j$ .*

**Definition 10.2.** *An Extension Database  $EXT_i$  is the only Elastic Resource  $ER$  that exists in the system. All Extension Databases are fully identical.*

*The Hihooi Autonomic Elasticity Model implements Hihooi's elasticity management strategy goals. Our strategic quality goal is to provide the maximum workload  $W$  scalability with the fewest possible elastic resources  $ER$ .*

### 10.1 The Hihooi Autonomic Elasticity Model

In this section we will analyze our model through an experiment. Our Model is supported via two important metrics (i) the Reads per Second (RPS) i.e., the total number of read responses divided by the total read time during the collection interval, and (ii) the Time per Call (TPC), which includes the time executing queries on the database and delivering the Results Sets, divided by the total number of calls during the collection interval. Since, Hihooi classifies SQL Statements into three affected classes i.e, RAS,CAS and TAS, the RPS and TPC metrics are specialized to the referencing class e.g., the RAS RPS and RAS TPC refer to RPS and TPC metric respectively, only for RAS Statements.

**Experiment Setup:** We create a draft benchmark based on a single table named *employees*. The table contains an arbitrary employees' data, indexed based on the *employee\_id*

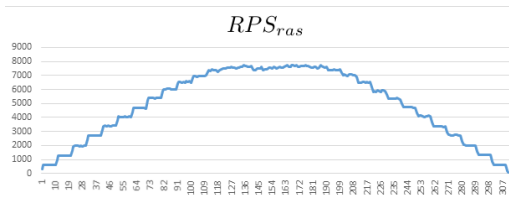


Figure 21: RAS Statements per second

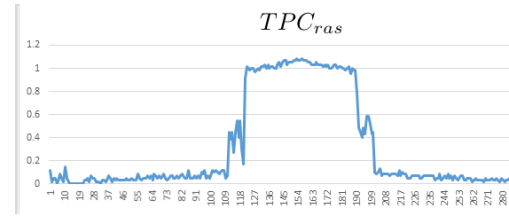


Figure 22: RAS Time per call

column, which is the table’s primary key. The benchmark is able to generate and dynamically mix all types of Hihooi’s affected SQL statements either in read or write mode. We run our Benchmark against Hihooi using only one Extension Database. For this experiment the *employees* table’s size was 10M rows. Every 5 minutes we start a new user. The experiment implies that every user repeatedly executes arbitrary RAS statements. The initial experiment uses RAS Statements, because RAS has fixed execution time due to the fact that behind the scene, the DBMS uses primary key index (typically implemented as a B+ tree). Therefore, overloading RAS Statements will immediately affect their execution time as well as, output. The experiment aims to increase the number of clients every five minutes until the maximum of 32 clients is reached. After that, we start the reverse process by stopping a user every 5 minutes, starting with the user that came first. During the experiment, we count the metrics  $RPS_{ras}$  and  $TPC_{ras}$  (metrics are defined in section 10.2) while the final results are demonstrated in figures 21 and 22.


**Experiment Results:** The  $RPS_{ras}$  output (Figure 21) shows that, when new users periodically enter into the system, the  $RPS_{ras}$  output follows a linear increase that gradually decreases, flattening the curve is an indication that the workload overloads the system (i.e., no resources are available to support the current load). Concurrently, the  $TPC_{ras}$  output (Figure 22) shows a execution time per call when new users periodically enter into the system. However, when the workload overloads the system, the  $TPC_{ras}$  output grows sharply. Finally, when the workload subsequently reduces the number of users (i.e., the load decrease),  $RPS_{ras}$  and  $TPC_{ras}$  outputs follow a downwards trend which is exactly the opposite from the initial.

**Experiment Findings:** From the experiment, it is illustrated that the overload detection can be achieved if we can quickly identify the stabilization/saturation of the RPS metric. The RPS saturation is due to the slow statements' execution on Extension Database, because, the current the Extension Database's resources can not handle the current load.

**How to identify  $RPS_{ras}$  stabilization:**

- (i) The Growth Rates Percentage ( $grp$ ) refer to the percentage change of a RPS within a specific time period e.g., between a period  $t$  (indicates *now*) and a previous period  $t-1$ . A period  $t$  changes when the number of users change. On each period change the system calculates  $RPS_{ma_t}$  moving average. As a result, a period's  $grp_t$  is calculated as

$$grp_t = \frac{RPS_{ma_t} - RPS_{ma_{t-1}}}{RPS_{ma_{t-1}}} \quad (1)$$

Stabilization is presented if the  $grp$  between two consecutive periods is  $\leq 0$ . The side picture, , illustrates how the  $grp$  is calculated during the experiment i.e., there is a spike when the new user enters the system, and later  $grp_t$  is stabilized one level up from the previews  $grp_{t-1}$ ; at that time  $grp_t$  are very closed to 0 but not  $\leq 0$ . Generally the  $grp$  metric is very sensitive and may give some incorrect indications but if used in conjunction with another measurement, namely, *TPC* it may be applied in this context.

- (ii) The ultimate approach is to measure the tangent angle between two RPS metrics (the past RPS and the current RPS). If the tangent angle is less than 30 degrees, then, RPS output is saturated. How to calculate the tangent angle measurement is described in Algorithm 10.1 which runs on the Transaction Manager.

**How to identify *TPC* slow performance:** The complexity of RAS, CAS and TAS statement are different. RAS statements are executed very fast (as they return a single row) compared to CAS (i.e., returns a number of table's columns for one or more table's rows) and TAS (that return all table's columns for all table rows). As a result, the *TPC* response

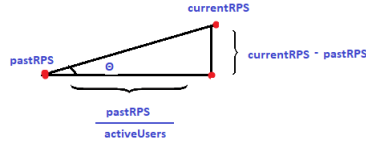
---

**Algorithm 10.1**  $IsRPS_{ras}$  Stabilized

---

**Require:**  $currentRPS_{ras}$ ,  $pastRPS_{ras}$ ,  $activeUsers$

- 1: **return**  $TRUE$  or  $FALSE$
- 2:  $activeUsers = activeUsers - 1$
- 3:  $x = \frac{currentRPS_{ras} - pastRPS_{ras}}{pastRPS_{ras}}$   
 $\frac{activeUsers}{activeUsers}$
- 4:  $radians = Math.atan(x)$
- 5: **IF** ( $Math.toDegrees(radians) < 30$ )
- 6: **return**  $TRUE$
- 7: **ELSE**
- 8: **return**  $FALSE$
- 9: **ENDIF**



10: *How to calculate tan angle :*

---

is depended from the variety of Affected Classes Statements that belong in the current workload. Fortunately, Hihooi is able to separate TPC to  $TPC_{ras}$ ,  $TPC_{cas}$  and  $TPC_{tas}$  and monitoring separately. As a result, the monitore these metrics of  $TPC_{ras}$  is enough to identify if the current TPC value has grown prohibitively. We identify TPC slow performance by treating it as an *Outlier Detection Problem* and solving it using Algorithm 10.2. For example, let  $tpc_1, tpc_2, tpc_3$  be the historical  $TPC_{ras}$  values calculated during the last three periods and  $tpc_4$  be the current  $TPC_{ras}$ . Based on Algorithm 10.2, the values a and b are evaluated as:

$$a = AVG((tpc_2 - tpc_1), (tpc_2 - tpc_3)) + 2 * (STDDEV((tpc_2 - tpc_1), (tpc_2 - tpc_3))) \quad (2)$$

$$b = (tpc_4 - tpc_3) \quad (3)$$

Based on the above framework, we can safely determine that any workload can overload our system if all conditions from definition 10.3 are apply.

**Definition 10.3.** We consider that a workload  $W$  overloads an Extension Database if the  $RPS_{ras}$  output is stabilized (i.e., Algorithm 10.1 returns  $TRUE$ ), causing a sharp increase of  $TPC_{ras}$  output (i.e., Algorithm 10.2 returns  $TRUE$ ).

---

**Algorithm 10.2** Is Current TPC an Outlier

---

**Require:** *currentTPC*

```
1: return TRUE or FALSE
2: [ ] tpcHistory // An ARRAY of  $TPC_{ras}$  values
3: [ ] tpcDiff = [tpcHistory.length - 1] // Temporary ARRAY
4: latestTpc = tpcHistory[tpcHistory.length - 1] // Last element of tpcHistory
5: k = 0
6: for(i = tpcHistory.length - 1 i >= 0 i--)
7:   if(i - 1 >= 0)
8:     diff = tpcHistory[i] - tpcHistory[i - 1]
9:     tpcDiff[k] = diff
10:    k ++
11:   endif
12: endfor
13: a = average(tpcDiff) + (2 * standardDeviation(tpcDiff))
14: b = currentTpc - latestTpc
15: if b > a
16:   return TRUE
17: else
18:   return FALSE
19: endif
```

---

**Evaluating Hihooi Autonomic Elasticity Model:** In order to evaluate the Hihooi Autonomic Elasticity Model we created a new experiment using the same workload and enabled Hihooi autonomic scale-out. Initially, we start Hihooi using one Extension DB. Every 5 minutes we start a new user. The experiment implies that every user repeatedly executes arbitrary *RAS* statements. We stopped the experiment when 32 users accessed the system. During the experiment, the system scales out two times, i.e., the experiment is completed with three Extension DB. The new resources enter the system after the tenth user and after the twentieth user. The experimental results (Figure 23) are very encouraging, because the total reads per second and total number of reads scale linearly.

## 10.2 Metrics Definitions

Hihooi collects every second a number of statistics and metrics. *Global* are the metrics collected by the Transaction Manager and *Local* metrics are collected by Extension DBs. The metrics that are related with Autonomic Elasticity are the following:

- (i)  $RAS_{q_{et}}$  is the query time for a *RAS* Statement (i.e., *elapsed time*) executed in the

Extension Database.  $RAS_{qet}$  includes the time  $t$  executing a query on database and delivering the results set  $rs$  to the *Transaction Manager*.

- (ii)  $CAS_{qet}$  is the query time for a CAS Statement (i.e., *elapsed time*) executed in the Extension Database.  $CAS_{qet}$  includes the time  $t$  executing a query on database and delivering the results set  $rs$  to the *Transaction Manager*.
- (iii)  $TAS_{qet}$  is the query time for a TAS Statement (i.e., *elapsed time*) executed in the Extension Database.  $TAS_{qet}$  includes the time  $t$  executing a query on database and delivering the results set  $rs$  to the *Transaction Manager*.
- (iv)  $RPS_{ras}$  is the *Total RAS Statements executed per second*.  $RPS_{ras}$  calculated from Transaction Manager (System wide metric) and from each active Extension Database (Local wide metric).
- (v)  $RPS_{cas}$  is the *Total CAS Statements executed per second*.  $RPS_{cas}$  calculated from Transaction Manager (System wide metric) and from each active Extension Database (Local wide metric).
- (vi)  $RPS_{tas}$  is the *Total TAS Statements executed per second*.  $RPS_{tas}$  calculated from Transaction Manager (System wide metric) and from each active Extension Database (Local wide metric).
- (vii)  $EXT_{RAS_{time}}$  is the cumulative elapsed time for all RAS statements  $q_{et}$  executed the last 1000ms. This metric is reset after its use. This metric is calculated in each Extension Database.

$$EXT_{RAS_{time}} = \sum RAS_{qet} \quad (4)$$

- (viii)  $EXT_{CAS_{time}}$  is the cumulative elapsed time for all CAS statements  $q_{et}$  executed the last 1000ms. This metric is reset after its use. This metric is calculated in each Extension Database.

$$EXT_{CAS_{time}} = \sum CAS_{qet} \quad (5)$$



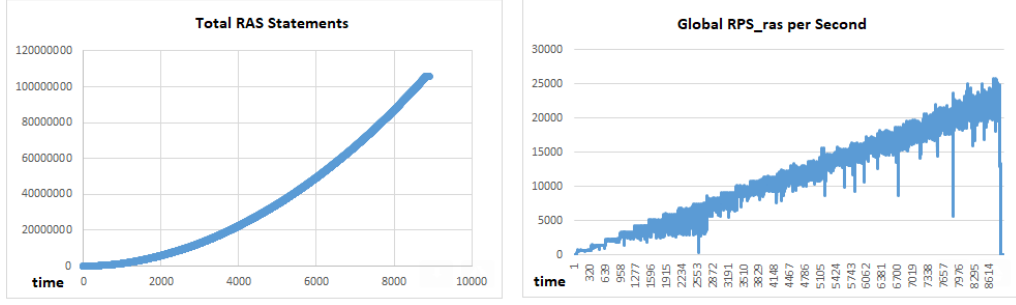


Figure 23: Results after enabling Hihooi Autonomic Elasticity

- (ix)  $EXT_{TAS_{time}}$  is the cumulative elapsed time for all CAS statements  $q_{et}$  executed the last 1000ms. This metric is reset after its use. This metric is calculated in each Extension Database.

$$EXT_{TAS_{time}} = \sum TAS_{q_{et}} \quad (6)$$

- (x)  $TPC_{ras}$  is the average *Time per call* for RAS statements.

$$TPC_{ras} = \frac{EXT_{RAS_{time}}}{RPS_{ras}} \quad (7)$$

- (xi)  $TPC_{cas}$  is the average *Time per call* for CAS statements.

$$TPC_{cas} = \frac{EXT_{CAS_{time}}}{RPS_{cas}} \quad (8)$$

- (xii)  $TPC_{tas}$  is the average *Time per call* for TAS statements.

$$TPC_{cas} = \frac{EXT_{TAS_{time}}}{RPS_{tas}} \quad (9)$$

### 10.3 Discussion

Using two experiments, we presented the initial work related to Hihooi Autonomic Elasticity Model. Through the experiment, we presented our model characteristics and parameters. Our strategic quality goal is to provide the maximum workload scalability, for any workload, with the fewest possible Extension Databases. The initial experimental results

shows that the Autonomic Hihooi Elasticity Model can potentially scale some workloads linearly.

## 11 Comparison with State-of-the-Art

Cloud computing technologies have made tremendous strides with respect to Database as a Service (DBaaS) offerings. Amazon, RackSpace, GoGrid, Google, Windows Azure SQL, Oracle Database Cloud Service and EnterpriseDB are all examples of pioneering database solutions in this field [10, 65, 42, 51, 3, 7, 69]. The companies that deploy these solutions, manage huge datacenters that run thousands of virtual machines. They employ traditional RDBMS, NoSQL and NewSQL data stores. Today, much attention is focused on the deployment of enterprise applications on current DBaaS technologies. A majority of these applications (making up 80% of all applications ) are not deployed on the cloud. The majority of these applications are relational and their transactions are dependent on ACID properties. The quality of these applications is high. Currently, these applications are deployed on traditional RDBMS such as Oracle, IBM DB2, MySQL, MS-SQL, Sysbase, Informix and PostgreSQL. The deployed Traditional RDBMS, that are able to support enterprise applications with scalability, are very expensive. The high cost of licensing, the expense of specialist hardware, and overprovisioned systems which must handle peak demands, are the main reasons for the high overheads. In addition, current solutions appear not to be truly elastic. Their scalability capabilities are static and cannot handle peak demands on the fly. Furthermore, the preponderance of current solutions present poor multi-tenancy. NoSQL and NewSQL solutions are not able to offer multi-tenancy, leading to misuse of resources and unpredictable performance result. As a consequence, the ability to offer multi-tenancy is fully dependent on IaaS's technology. Also, none of the current solutions offer "Quality of Service" (QoS) with respect of DBaaS's performance, and no solution protects DBaaS's performance through a "Service Level Agreement" (SLA) contract.

Database replication comes in two forms: (i) *master-slave*, where one primary copy handles all writes and the other replicas process only reads [77, 73, 104]; and (ii) *multi-master*, where all replicas serve both reads and writes [27, 55, 41]. Each form can be implemented either inside the database kernel or outside in a middleware layer. While the former approach provides opportunities for various optimizations and a tight coupling of

concurrency and replica control, it is heavily invasive and database-engine specific [57]. The middleware approach, also employed by Hihooi, leads to a seamless separation of concerns, supports unmodified database systems and applications, and can enable heterogeneous environments.

Postgres-R [55] was one of the first multi-master replication systems to use group communication primitives with strong ordering to enable scalability and 1-copy-serializability, while a later version offered snapshot isolation (SI) [110]. Middle-R [73] was the middleware extension of Postgres-R that moved group communication outside the database engine but still required database modifications for extracting and applying tuple-based updates. Other similar systems that rely on group communication primitives and offer SI are Tashkent [41] and SI-Rep [60]. C-JDBC [27] is also a multi-master middleware system but does not require database modifications as it uses JDBC drivers like Hihooi. The system offers consistency guarantees through table-level locking at the middleware level.

DBFarm [81] builds upon ideas from Middle-R (and thus requires database engine modifications) but offers a master-slave middleware system. As such, a read transaction is delegated to some replica but it is blocked until that replica is consistent with the primary. [11, 54] present middleware solutions that require a predeclaration of the access pattern of all transactions to enable efficient scheduling. In [104], the middleware will first execute a write transaction on the primary replica, extract lock-based concurrency information, and use that to enforce a transaction scheduling to the replicas, which prevents conflicting schedules. Unlike Hihooi, [104] requires the underlying databases to use strict two-phase locking and cannot handle snapshot isolation, which is now widely used. Ganymed [77] is a similar 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 read/write sets to find the replicas, including the Primary DB, that are consistent for each read transaction to run on. In doing so, Hihooi is the first replication-based middleware to offer such fine-grained statement-based routing, even within multi-statement write transactions. Pgpool-II [63] is

another PostgreSQL-specific replication middleware solution that ships and applies WAL entries to the replicas. Pgpool-II, similar to DBFarm and Ganymed, apply all database modifications serially at the replicas, as opposed to Hihooi that applies them in parallel.

Another way in which 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 [60, 26]. The Transactions Buffer acts as a highly available propagation medium for all database modifications that need to be applied asynchronously to active replicas, improving network load distribution and simplifying recovery procedures. Amazon Aurora [106] has a different architecture that decouples compute from storage while employing primary copy replication to achieve read scale-out. Aurora uses physical replication, where the redo log records are replayed in the replicas, allowing them to be physically identical to the primary. Such an approach, however, cannot be used to scale existing single-node databases (unlike Hihooi).

Other systems such as Hyder [19, 18] and Tango [15] provide the abstraction of a replicated in-memory data structure backed by a shared log, and leverage the shared log to enable fast transactions across different objects. [108] and [83] provide log shipping from a primary copy. The former uses synchronous writes so it avoids concurrency issues from reading from replicas, but it relies on the presence of InfiniBand and NVRAM to be efficient. The latter replays logs at the level of records but the approach only targets the scenario of primary-backup replication with a single backup instead of multiple replicas. KuaFu [111] is a primary-backup, row-based replication system that 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 read/write 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.

Commercial clustering solutions such as Oracle RAC [70] and IBM DB2 pureScale [52] rely on the use of specialized hardware and network-attached storage to work. Hence,

unlike our approach, the system cannot easily be installed on a set of commodity servers. Finally, other database replication products such as Oracle Golden Gate [48] exist, but only offer weak consistent properties and are meant to be used for off-line reporting or disaster recovery plans.

Data partitioning is another popular scale-out approach that partitions and distributes data across cluster nodes [24, 99]. Such approaches are amenable to dynamic scaling via migrating data to existing or new nodes in order to diminish performance issues due to skew or heavy loads. Accordion [90] migrates data at a coarse predefined granularity, whereas E-Store [98] and Clay [91] work at a finer tuple-level granularity. The aforementioned approaches perform data migrations after detecting performance issues, whereas P-Store [97], another elastic OLTP DBMS, focuses on workload prediction and proactive migration. One of the key scalability hurdles of data partitioning approaches are transactions spanning multiple partitions as they require locking or other specialized protocols; a non-existent issue for Hihooi as all transactions have access to the full database. Finally, the issue of dynamic scaling is orthogonal to our approach and something we plan to work on in the near future.

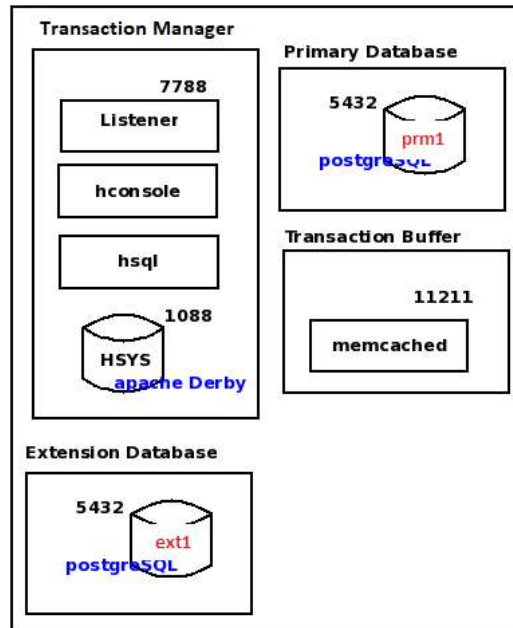
## 12 Conclusions

The rapid growth of workloads in combination with a variability of connected users often cause database management systems to become overloaded. This problem was the reason for designing new systems or transforming existing systems to be able to offer both workload scalability and elasticity. From the outset of our system design, we have assumed that true elasticity of system's scalability could not be realized if there was no autonomy in the system. As a result, the design of the system was based on the idea of having the basis of self management of the system. Initially, as a replication-based middleware, Hihooi is able to provide workload scalability to existing databases without sacrificing consistency. In addition, Hihooi's parallel replication algorithm allows the Extension DBs to reach consistency quicker. Additionally, Hihooi's routing algorithms avoid any delays by routing read statements to consistent replicas. All the above contributions are extensively evaluated showcasing that the workload scalability is attainable with Hihooi. Finally, the initial experiments on the Hihooi Elasticity Model show that model is able to achieves a workload's goals by moderating elasticity resources automatically. The system is able to add and remove Extension DBs online and re-balances online the system load. Also, the workload's classification to Statements Affecting Classes, provides the perspective of generalizing the elasticity requirements of any workload by helping the system to recognize more easily saturation in performing the workload. Hence, Hihooi can jump start interesting research towards automated elasticity as well as the creation of new cloud-based offerings.

# Appendix

## Hihooi Installation and Configuration Guide Version 1.0 (Rhino Guard)

This document describes how to install Hihooi on windows standalone machine.



Hihooi's Components and TCP/IP Ports Distribution



Before continue with Hihooi installation please read the document "Installing Hihooi Depended Software for windows" and ensure that all components are installed as described in the document.

### Configuration Steps :

1. Create the Primary and Extension Databases on localhost
2. Install and Configure Transaction Manager
  1. Clone Software
  2. Init installation
  3. Create executable scripts and compile the Transaction Manager code
  4. Create, Configure and Start hsys repository
  5. Orchestrate Transaction Manager using hconsole
  6. Start Transaction Manager



3. Install and Configure Extractor
  1. Create executable scrips and compile the Extactor code
  2. Configure Local Extractor

## Install and Configure Transaction Manager

### Clone Software

```
Create HIHOOI_BASE
-----

cd \
mkdir Hihooi-Rhino-Guard
cd Hihooi-Rhino-Guard
set HIHOOI_BASE=C:\Hihooi-Rhino-Guard

Clone Software
-----

git clone -b rhino-guard-02-12-2017 mgeorgiou@dicl.cut.ac.cy:/home1/git/hihooi/hihooi.git
cd hihooi

Create HIHOOI_HOME
-----

set HIHOOI_HOME=C:\Hihooi-Rhino-Guard\hihooi
```

Line 1 : Create the base hihooi directory e.g., Hihooi-0.7

Line 2 : Go in directory Hihooi-07

Line 3: Clone software using the git command.

The clone command creates the directory hihooi which contains the directories extractor (for extractor), hih-client (client software) and lis (Transaction Manager).

Line 3: Set the session enviroment variable HIHOOI\_HOME

### Init installation

```
1. cd %HIHOOI_HOME%
2. init.bat
```

Line 1: Go in %HIHOOI\_HOME%

Line 2: Initialize the Transaction Manager installation using the init.bat command which creates the appropriate directories and configuration files.

## Create executable scripts and compile the Transaction Manager code

```
1. cd %HIHOOI_HOME%\lis\conf
2. vi hihooi.conf
```

Line 1: Go in the Transaction Manager's configuration directory

Line 2: Edit the hihooi.conf file using your familiar editor

According the example architecture illustrated in figure 1 , add the following mandatory fields in the hihooi.conf , save and exit from editor .



**Replace only the JAVA\_HOME path**



**Open a notepad as administrator and add the following entry in**

C:\Windows\System32\drivers\etc\hosts

```
127.0.0.1          ext01
```

Where ext is the extractor\_name and 01 the extractor\_id



**Please to do not change the default value for the property  
SERVICE\_DEFAULT\_DB\_VENDOR=PostgreSQL 9.3.6**

```
JAVA_HOME="C:\Program Files\Java\jdk1.8.0_144"
EXTRACTOR_ID=01
EXTRACTOR_NAME=ext
EXTENSION_DB_NAME=ext1
EXTENSION_DB_USERNAME=postgres
EXTENSION_DB_PASSWORD=postgres
SERVICE_REPOSITORY_HOST=localhost
SERVICE_REPOSITORY_PORT=1088
SERVICE_REPOSITORY_DB_NAME=/hihsrv
SERVICE_NAME=TESTSRV
SERVICE_DEFAULT_DB_VENDOR=PostgreSQL 9.3.6
SERVICE_DEFAULT_DB_PORT=5432
SERVICE_DEFAULT_DB_MAX_CONNECTIONS=10
SERVICE_TRANSACTION_MANAGER_HOST=localhost
SERVICE_TRANSACTION_MANAGER_PORT=7777
SERVICE_TRANSACTION_MANAGER_IP=127.0.0.1
SERVICE_MEMCACHED_HOST=localhost
SERVICE_MEMCACHED_PORT=11211
SERVICE_MEMCACHED_IP=127.0.0.1
SERVICE_MEMCACHED_INSTANCE_NAME=MEM1
SERVICE_MEMCACHED_INIT_CONNECTIONS=1
SERVICE_MEMCACHED_MAX_CONNECTIONS=1
SERVICE_MEMCACHED_MIN_CONNECTIONS=1
Transet_Management=serial
```

```
dirty_Objects_Mode=Mix
poolConnections=10
parallelApplier=false
transetGroupApplier=false
serialThreads=4
databasePropertyFile=database.properties.txt
listeningIp=localhost
VERSION="Rhino Guard"
```

```
3. %HIHOOI_HOME%\lis\installer.bat
```

Line 3: run the Transaction Manager's installer . The installer compiles the transaction manager source code, and creates the appropriate executable scripts under the %HIHOOI\_HOME%\lis\bin directory.

<b>hconsole</b>	Start hconsole tool
<b>hsql</b>	Start hsql console
<b>ij</b>	Start ij console
<b>jcompile</b>	The script that compiles the Hihooi Transaction Manager code
<b>shutDerby</b>	The script that shuts the hsys Transaction Manager
<b>startDerby</b>	The script that starts the hsys Transaction Manager
<b>startLis</b>	The script that starts the Transaction Manager.

Table 1: Transaction Manager command

## Create, Configure and Start hsys repository

### Start HSYS Transaction Manager

HSYS is located in an Apache Derby Database, thus in order to startup and connect with the database you need to start the database's Transaction Managers. The HSYS Transaction Manager is started on tcp port as configured in SERVICE\_REPOSITORY\_PORT parameter and hosted as configured the SERVICE\_REPOSITORY\_HOST parameter.

Start HSYS Transaction Manager using the command:

```
1. %HIHOOI_HOME%\lis\bin\startDerby.bat
```

**Output:**

```
Apache Derby Network Server - 10.10.2.0 - (1582446) started and
ready to accept connections on port 1088
```

**Important**  
**\*\*\* Leave the window open**

### Optional : Create JDerby as Windows service manageable by windows services

```
nssm install JDerby C:\Hihooi-Rhino\hihooi\lis\bin\startDerby.bat
```

### Create HSYS database objects

In order to create the HSYS database you need to connect with Transaction Manager using the ij tool. When you connect with ij connect/create the HSYS database named as hihsrv and run the script hsys.sql which creates the HSYS's database objects.

Open a new CMD terminal and run the following:

```
1. $HIHOOI_HOME/lis/bin/ij.bat
2. connect
'jdbc:derby://SERVICE_REPOSITORY_HOST:SERVICE_REPOSITORY_PORT/hihsrv;create=
true';
3. ij> run 'HIHOOI_HOME_PATH\lis\db\hsys.sql' ;
4. ij> exit;
```

#### Example:

```
ij>connect 'jdbc:derby://localhost:1088/hihsrv;create=true';
ij> run 'C:\Hihooi-Rhino-Guard\hihooi\lis\db\hsys.sql' ;
ij>exit;
```

Line 1: Run the ij tool

Line 2: Create the hihsrv database if not exist . Please replace the SERVICE\_REPOSITORY\_HOST and SERVICE\_REPOSITORY\_HOST parameters with the values already saved in hihooi.conf.

Line 3: Create the HSYS database objects by running the hsys.sql script. Please replace HIHOOI\_HOME\_PATH with the physical path.

Line 4: Exit from ij tool.

### Orchestrate Hihooi components using hconsole



For postgresQL 9.6 and 10.1 ( supported version for windows 10 ) copy and paste the following jar file

```
C:\Hihooi-Rhino-Guard\hihooi\drivers\postgresql-42.1.4.jar into
C:\Hihooi-Rhino-Guard\hihooi\lis\lib
```

Line 1: Start the hconsole.

```
1. %HIHOOI_HOME%\lis\bin\hconsole.bat
```



A full description of **Hconsole** can be founded in Hconsole.docx document.

Initially the hconsole creates a system service (if the service does not exist).  
On first login, the hconsole responses the following output :

```
Working on Service TESTSRV Service id 1  
Working on checkpoint using the Primary DB
```

Hihooi's service is unique and contains one transaction manager, one transaction buffer (memcached), one primary database and at least one or more extension databases.

```
2. add Transaction Manager;  
3. add memcached;
```

**Example output:**

```
hconsole>add Transaction Manager;  
Transaction Manager localhost added.  
hconsole>add memcached;  
Memcached MEM1 added.  
log4j:WARN No appenders could be found for logger (hih.MemcachedClient).  
log4j:WARN Please initialize the log4j system properly.  
  
*** Successfully ADD key:1234567890,text:This a test  
*** Successfully GET key:1234567890,text:This a test ..  
  
0  
hconsole>
```

Line 2: Add Transaction Manager according the properties in hihooi.conf

```
SERVICE_TRANSACTION_MANAGER_HOST=localhost  
SERVICE_TRANSACTION_MANAGER_PORT=7777  
SERVICE_TRANSACTION_MANAGER_IP=127.0.0.1
```

Line 3: Add Tranaction Buffer ( memcached) according the properties in hihooi.conf

```
SERVICE_MEMCACHED_HOST=localhost  
SERVICE_MEMCACHED_PORT=11211  
SERVICE_MEMCACHED_IP=127.0.0.1  
SERVICE_MEMCACHED_INSTANCE_NAME=MEM1  
SERVICE_MEMCACHED_INIT_CONNECTIONS=1  
SERVICE_MEMCACHED_MAX_CONNECTIONS=1  
SERVICE_MEMCACHED_MIN_CONNECTIONS=1
```

```
4. add primary database host localhost name prm1 username postgres
password postgres;

5. add extension database host ext01 name ext1 username postgres
password postgres;
```

Line 4 : add the primary database by providing database end points.



**The adding database must be open and running because the system tries to test the database connection.**

**You can add all your available databases from different experiments**

**Related Hconsole's commands associated with database**

- show database;
- show database schema;
- ping database <connector\_id>;

To enable database replication from primary to extension database you need create a checkpoint between hihooi and primary database.

```
hconsole>show database schema;
+-----+-----+-----+
|CONNECTOR_ID|DB_NAME          |DB_USER          |
+-----+-----+-----+
|1           |prm1             |postgres         |
+-----+-----+-----+
|4           |ext1             |postgres         |
+-----+-----+-----+
```

```
6. create checkpoint using primary database <connector_id>;
```

**Example:** create checkpoint using primary database 1;

Line 6: Create the service checkpoint using the given connector id primary db.



The Hihooi checkpoint mechanism is used in order for a system to have always a consistent starting point with Primary DB. Hihooi checkpoint is based on DBMS's numbering system technique in order to identify which are the last, consistent and visible transaction in their system. The system starts a new Checkpoint when there is inconsistency between Primary DB and the system. This can happen on data changes occurring out of the system e.g, bulk loads. The Checkpoint creation includes the creation of a new SEED Database or SEED Backup where new Extension DBs will be re-created based on that instance. In order to change the Primary DB for a specific runtime you need to cancel the current checkpoint .

Related Hconsole command related with checkpoint are:

- cancel checkpoint;
- show checkpoint;
- show checkpoint member;
- remove database member database <connector\_id i>,..*<connector\_id n>*

```
7. add checkpoint member database <connector id>;
```

**Example:** `hconsole>add checkpoint member database 4;`

Line 7: Add checkpoint member for the already existing extensions databases

## Start Transaction Manager

Start the Transaction Manager using the command

```
1. %HIHOOI_HOME%\lib\bin\startLis.bat
```

Successful Transaction Manager output:

```
...  
Hihooi Transaction Manager Version 0.6.2 mantis  
Started...Listen on 172.16.56.48 at 7788
```

## Query the database using hsql

```
1. %HIHOOI_HOME%\lis\bin\hsql.bat  
2. Connect to:localhost:7788@TESTSRV  
3. hsql>connect;  
4. <row>  
5. <output>Session 4016ed39-67d5-49 is created.</output>  
6. <\row>  
7. hsql>insert into dual values (1);  
8. <row>  
9. <output>affected rows 1</output>  
10. <\row>  
11. hsql>select count(*) from dual;  
12. 1.  
13. hsql>\q
```

Line 1 : Start the hsql

Line 3: Execute connect to connect with the system

**Hsql commands are described in hsql document**



## Shutdown Transaction Manager

The Transaction Manager should be shutdown after it use. The shutdown operation saves the current state of the system.

```
%HIHOOI_HOME%/lis/bin/hsql.bat
Connect to:localhost:7788@TESTSRV
hsql>connect;
hsql>shutdown;
```

If you close the Transaction Manager abnormally , you need to restart memcached , and set the extension's transet\_id to Transaction Manager's transet\_id -1 ( checkpoint.conf) file ( see next chapter).

## Install and Configure Extractor

### Create executable scripts and compile the Extractor code

```
1. cd %HIHOOI_HOME%\extractor\conf
2. vi hihooi.conf
** copy hihooi.conf from %HIHOOI_HOME%\lis\conf if extractor and Transaction Manager are hosted in
the same machine
```

Line 1: Go in extractor configuration directory

Line 2: Edit the hihooi.conf file using your familiar editor

```
3. %HIHOOI_HOME%\extractor\installer.bat
```

Line 3: run the extractor installer.bat . The installer compiles the extractor source code, and creates the appropriate executable scripts under the %HIHOOI\_HOME%\extractor\bin directory.

<b>./startExtractor</b>	Start extractor
<b>./showTranset</b>	Set current transet id
<b>jcompile</b>	The script that compiles the Hihooi Extractor code

Table 2: Extractor command



## Start Extractor

```
4.  $HIHOOI_HOME/extractor/bin/showTranset.bat
    1.8999
5.  $HIHOOI_HOME/extractor/bin/startExtractor.bat
```

Line 4: Verify the setTranset Command or view the current Transet id

Line 5: Start The extractor.

## HConsole

HConsole is an interactive console application that can be used for configuring and managing Hihooi, including adding/removing replicas, creating checkpoints. HConsole save system's configuration in HSYS database, thus, a connection with hsys database is mandatory. However for adding/removing replicas, Hconsole is communicated with Transaction Manager in order to notify him of the changes that will be made to the system.

## Start HConsole

```
%HIHOOI_HOME/lis/startHconsole.bat
```

## Operations

```
show service
```

```
    database [schema]
    Transaction Manager
    memcached
    checkpoint [member]
```

```
add service
```

```
    Transaction Manager
    primary database
    extension database
    add memcached
    add checkpoint member database <database_id_i,..,database_id_n>
```

```
set default service <service_name>
```

```
create checkpoint using primary database <primary_database_id>
```

```
cancel checkpoint
```

```
remove checkpoint member database <database_id_i,...,database_id_n>  
    database <database_id_i,...,database_id_n>
```

```
ping database <database_id>
```

## HSQL

HSQL is an interactive console application that can be used for execute queries and monitoring Hihooi. HSQL is connected directly with Hihooi using the hihooi API.

### Start HConsole

```
%HIHOOI_HOME/lis/hsq1.bat
```

### Operations

```
connect
```

```
select <sql>
```

```
update <sql>
```

```
delete <sql>
```

```
drop <sql>
```

```
create <sql>
```

```
alter <sql>
```

```
truncate <sql>
```

```
commit <sql>
```

```
rollback <sql>
```

```
start transaction
```

```
set consistency level [1|2|3|4|5]
```

```
disconnect
```

```
Transaction Manager
```

```
print connector
```

```
    database
```

```
    dobj
```

refresh connectors

## References

- [1] Apache h-base, 2014.
- [2] Apache hadoop, 2014.
- [3] Gogrid, 2014.
- [4] High availability, load balancing, and replication, February 2014.
- [5] Innodb , storage engine, 2014.
- [6] Peer-to-peer transactional replication, February 2014.
- [7] Rackspace, 2014.
- [8] Atul Adya, Barbara Liskov, and Patrick O’Neil. Generalized Isolation Level Definitions. In *Proc. of ICDE*, pages 67–78. IEEE, 2000.
- [9] Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering*, pages 576–585. IEEE, 2008.
- [10] Amazon. Amazon relational database service (amazon rds), February 2014.
- [11] Cristiana Amza et al. Distributed Versioning: Consistent Replication for Scaling Back-End Databases of Dynamic Content Web Sites. In *Proc. of MIDDLEWARE*, pages 282–304. Springer-Verlag, 2003.
- [12] Apache. Apache derby.
- [13] Shivnath Babu and Herodotos Herodotou. Massively Parallel Databases and MapReduce Systems. *FnTDB*, 5(1):1–104, 2013.
- [14] Jason Baker et al. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, pages 223–234, 2011.
- [15] Mahesh Balakrishnan et al. Tango: Distributed Data Structures over a Shared Log. In *Proc. of SOSR*, pages 325–340. ACM, 2013.

- [16] Hal Berenson, Phil Bernstein, Jim Gray, et al. A Critique of ANSI SQL Isolation Levels. In *Proc. of SIGMOD*, pages 1–10. ACM, 1995.
- [17] David Bermbach and Jörn Kuhlentkamp. Consistency in distributed storage systems. In *Networked Systems*, pages 175–189. Springer, 2013.
- [18] Philip A Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing Optimistic Concurrency Control for Tree-structured, Log-structured Databases. In *Proc. of SIGMOD*, pages 1295–1309. ACM, 2015.
- [19] Philip A Bernstein et al. Hyder-A Transactional Record Manager for Shared Flash. In *Proc. of CIDR*, volume 11, pages 9–20, 2011.
- [20] Philip A Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.
- [21] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00)*, pages 7–10, NY, July 16–19 2000. ACM Press.
- [22] Emiliano Casalicchio and Luca Silvestri. Mechanisms for sla provisioning in cloud-based service providers. *Computer Networks*, 57(3):795–810, 2013.
- [23] Rick Cattell. Scalable SQL and noSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.
- [24] Rick Cattell. Scalable SQL and NoSQL Data Stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.
- [25] Emmanuel Cecchet. C-JDBC: a middleware framework for database clustering. *IEEE Data Eng. Bull.*, 27(2):19–26, 2004.
- [26] Emmanuel Cecchet, George Candea, and Anastasia Ailamaki. Middleware-based Database Replication: The Gaps between Theory and Practice. In *Proc. of SIGMOD*, pages 739–752. ACM, 2008.

- [27] Emmanuel Cecchet et al. C-JDBC: Flexible Database Clustering Middleware. In *Proc. of USENIX ATC*, pages 9–18. USENIX, 2004.
- [28] Emmanuel Cecchet, Marguerite Julie, and Willy Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX Annual Technical Conference*, number CONF, 2004.
- [29] Kristina Chodorow. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. O’Reilly Media, Inc., 2013.
- [30] Clustrix. Clustrix scale out newsq database in the cloud, 2014.
- [31] Richard Cole, Florian Funke, et al. The Mixed Workload CH-benCHmark. In *Proc. of DBTest Workshop*, pages 8:1–8:6. ACM, 2011.
- [32] Thomas M. Connolly and Carolyn E. Begg. *Database Systems : A Practical Approach to Design, Implementation, and Management*. Addison-Wesley Pub Co, 1999. Harold Cohen Library Liverpool.
- [33] Brian F. Cooper et al. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of SoCC*, pages 143–154. ACM, 2010.
- [34] James C Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Google’s Globally Distributed Database. *ACM TOCS*, 31(3):8, 2013.
- [35] James C Corbett, Jeffrey Dean, Michael Epstein, et al. Spanner: Google’s Globally Distributed Database. *ACM TOCS*, 31(3):8, 2013.
- [36] Carlo Curino et al. Workload-aware Database Monitoring and Consolidation. In *Proc. of SIGMOD*, pages 313–324. ACM, 2011.
- [37] Khuzaima Daudjee and Kenneth Salem. Lazy Database Replication with Snapshot Isolation. In *Proc. of VLDB*, pages 715–726. VLDB End., 2006.
- [38] Cristian Diaconu et al. Hekaton: SQL Server’s Memory-optimized OLTP Engine. In *Proc. of SIGMOD*, pages 1243–1254. ACM, 2013.

- [39] Djellel Eddine Difallah et al. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB Endowment*, 7(4):277–288, 2013.
- [40] Jennie Duggan et al. Performance Prediction for Concurrent Database Workloads. In *Proc. of SIGMOD*, pages 337–348. ACM, 2011.
- [41] Sameh Elnikety et al. Tashkent: Uniting Durability with Transaction Ordering for High-performance Scalable Database Replication. *ACM SIGOPS Review*, 40(4):117–130, 2006.
- [42] EnterpriseDB. Enterpisedb - the postgres database, February 2014.
- [43] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Hyperdex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.
- [44] Brad Fitzpatrick. Distributed Caching with Memcached. *Linux Journal*, 2004(124):5, 2004.
- [45] Guilherme Galante and Luis Carlos E de Bona. A survey on cloud computing elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pages 263–270. IEEE Computer Society, 2012.
- [46] Jim Gray, Pat Helland, et al. The Dangers of Replication and a Solution. *ACM SIGMOD Record*, 25(2):173–182, 1996.
- [47] Katarina Grolinger et al. Data Management in Cloud Environments: NoSQL and NewSQL Data Stores. *JoCCASA*, 2(1), 2013.
- [48] Ankur Gupta. *Oracle Goldengate 11g Complete Cookbook*. Packt Publishing, 2013.
- [49] Ibrahim Abaker Targio Hashem, Ibrar Yaqoob, Nor Badrul Anuar, et al. The Rise of Big Data on Cloud Computing: Review and Open Research Issues. *Information Systems*, 47:98–115, 2015.

- [50] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing ({ICAC} 13)*, pages 23–27, 2013.
- [51] Inc Heroku. Heroku-postgres, February 2014.
- [52] IBM DB2 pureScale, 2014. <http://www-01.ibm.com/software/data/db2/linux-unix-windows/purescale/>.
- [53] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a consumer can measure elasticity for cloud platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 85–96, 2012.
- [54] Ricardo Jiménez-Peris, Marta Patiño-Martínez, Bettina Kemme, and Gustavo Alonso. Improving the Scalability of Fault-Tolerant Database Clusters. In *Proc. of ICDCS*, pages 477–484. IEEE, 2002.
- [55] Bettina Kemme and Gustavo Alonso. Don'T Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *Proc. of VLDB*, pages 134–143. Morgan Kaufmann Publishers, 2000.
- [56] Bettina Kemme and Gustavo Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems (TODS)*, 25(3):333–379, 2000.
- [57] Bettina Kemme and Gustavo Alonso. Database Replication: A Tale of Research Across Communities. *PVLDB*, 3(1):5–12, 2010.
- [58] Tirthankar Lahiri, Vinay Srihari, Wilson Chan, Neil Macnaughton, and Sashikanth Chandrasekaran. Cache fusion: Extending shared-disk clusters with shared caches. In *VLDB*, volume 1, pages 683–686, 2001.
- [59] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Review*, 44(2):35–40, 2010.



- [60] Yi Lin et al. Middleware Based Data Replication Providing Snapshot Isolation. In *Proc. of SIGMOD*, pages 419–430. ACM, 2005.
- [61] Karan B Maniar and Chintan B Khatri. Data science: Bigtable, mapreduce and google file system. *International Journal of Computer Trends and Technology (IJCTT)*, 16(03):115–118, 2014.
- [62] Ludovic Marcotte. Database Replication with Slony-I. *Linux Journal*, 2005(134):1, 2005.
- [63] Jayadevan Maymala. *PostgreSQL for Data Architects*. Packt Publishing, 2015.
- [64] MemSQL. *MemSQL: The Database for Real-time Applications*, 2018.
- [65] Microsoft. Windows azure, February 2014.
- [66] Toshimi Minoura. Multi-level concurrency control of a database system. In *Proceedings Fourth Symposium on Reliability in Distributed Software and Database Systems (4th SRDS'84)*, pages 156–168, Silver Spring, MD, USA, October 1984. IEEE Computer Society Press.
- [67] MySQL. The myisam storage engine, 2014.
- [68] Amro Najjar, Xavier Serpaggi, Christophe Gravier, and Olivier Boissier. Survey of elasticity management solutions in cloud computing. In *Continued Rise of the Cloud*, pages 235–263. Springer, 2014.
- [69] Oracle. Your oracle database in the cloud, February 2014.
- [70] Oracle Real Application Cluster, March 2017. <http://www.oracle.com/technetwork/database/options/clustering/>.
- [71] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*, 2011.
- [72] Esther Pacitti. *Improving data freshness in replicated databases*. PhD thesis, INRIA, 1999.

- [73] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent Database Replication at the Middleware Level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.
- [74] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.
- [75] Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems (TOCS)*, 23(4):375–423, 2005.
- [76] R. Peterson and J. Strickland. Log write-ahead protocols and IMS/VS logging. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Sys.*, page 216, Atlanta, GA, March 1983.
- [77] Christian Plattner and Gustavo Alonso. Ganymed: Scalable Replication for Transactional Web Applications. In *Proc. of MIDDLEWARE*, pages 155–174. Springer-Verlag, 2004.
- [78] Christian Plattner and Gustavo Alonso. Ganymed: Scalable replication for transactional web applications. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 155–174. Springer, 2004.
- [79] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. DBFarm: A scalable cluster for multiple databases. In Maarten van Steen and Michi Henning, editors, *Middleware 2006, ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, November 27-December 1, 2006, Proceedings*, volume 4290 of *Lecture Notes in Computer Science*, pages 180–200. Springer, 2006.
- [80] Christian Plattner, Gustavo Alonso, and M Tamer Özsu. Dbfarm: A scalable cluster for multiple databases. In *ACM/IFIP/USENIX International Conference on*

- Distributed Systems Platforms and Open Distributed Processing*, pages 180–200. Springer, 2006.
- [81] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. DBFarm: A Scalable Cluster for Multiple Databases. In *Proc. of MIDDLEWARE*, volume 4290, pages 180–200. Springer, 2006.
- [82] Dai Qin et al. Scalable Replay-Based Replication For Fast Databases. *PVLDB Endowment*, 10:2025–2036, 2017.
- [83] Dai Qin et al. Scalable Replay-based Replication for Fast Databases. *PVLDB*, 10(13):2025–2036, 2017.
- [84] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using paxos to build a scalable, consistent, and highly available datastore. *CoRR*, abs/1103.2408, 2011.
- [85] D. Reed. Naming and synchronization in a decentralized computer system. *Ph.D. Thesis*, 1978.
- [86] Sasko Ristov, Radu Prodan, Marjan Gusev, Dana Petcu, and Jorge Barbosa. Elastic cloud services compliance with gustafson’s and amdahl’s laws. 2016.
- [87] Mikael Ronstrom and Lars Thalmann. MySQL Cluster Architecture Overview. Technical report, MySQL, 2014.
- [88] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 500–507. IEEE, 2011.
- [89] Nilabja Roy, Abhishek Dubey, Aniruddha Gokhale, and Larry Dowdy. A capacity planning process for performance assurance of component-based distributed systems. *ACM SIGSOFT Software Engineering Notes*, 36(5):41–41, 2011.
- [90] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, et al. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *PVLDB*, 7(12):1035–1046, 2014.

- [91] Marco Serafini, Rebecca Taft, Aaron J Elmore, Andrew Pavlo, et al. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *PVLDB*, 10(4):445–456, 2016.
- [92] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *2011 31st International Conference on Distributed Computing Systems*, pages 559–570. IEEE, 2011.
- [93] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 5. ACM, 2011.
- [94] Gokul Soundararajan, Cristiana Amza, and Ashvin Goel. Database replication policies for dynamic content applications. In Yolande Berbers and Willy Zwaenepoel, editors, *EuroSys*, pages 89–102. ACM, 2006.
- [95] Michael Stonebraker. SQL databases v. noSQL databases. *Commun. ACM*, 53(4):10–11, 2010.
- [96] Michael Stonebraker and Ariel Weisberg. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bulletin*, 36(2):21–27, 2013.
- [97] Rebecca Taft, Nosayba El-Sayed, Marco Serafini, Yu Lu, Ashraf Aboulnaga, Michael Stonebraker, et al. P-Store: An Elastic Database System with Predictive Provisioning. In *Proc. of SIGMOD*, pages 205–219. ACM, 2018.
- [98] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, et al. E-Store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *PVLDB*, 8(3):245–256, 2014.
- [99] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, et al. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. of SIGMOD*, pages 1–12. ACM, 2012.
- [100] TokuDB. Tokudb: Mysql performance, mariadb performance.

- [101] TPC-C Benchmark, Revision 5.11.0, 2010. [www.tpc.org/tpcc/](http://www.tpc.org/tpcc/).
- [102] TPC-H Benchmark, Revision 2.17.3, 2017. [www.tpc.org/tpch/](http://www.tpc.org/tpch/).
- [103] Bhuvan Uргаonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile dynamic provisioning of multi-tier internet applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 3(1):1, 2008.
- [104] Ben Vandiver et al. Tolerating Byzantine Faults in Transaction Processing Systems using Commit Barrier Scheduling. *ACM SIGOPS Review*, 41(6):59–72, 2007.
- [105] Luis M Vaquero, Luis Roderо-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.
- [106] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, et al. Amazon Aurora: Design Considerations for High Throughput Cloud-native Relational Databases. In *Proc. of SIGMOD*, pages 1041–1052. ACM, 2017.
- [107] VoltDB. Voltdb , in-memory database and analytic.
- [108] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. Query Fresh: Log Shipping on Steroids. *PVLDB*, 11(4):406–419, 2017.
- [109] Bill Wilder. *Cloud Architecture Patterns*. O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., 2012-09-20.
- [110] Shuqing Wu and Bettina Kemme. Postgres-R (SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *Proc. of ICDE*, pages 422–433. IEEE, 2005.
- [111] Mao Yang et al. KuaFu: Closing the Parallelism Gap in Database Replication. In *Proc. of ICDE*, pages 1186–1195. IEEE, 2013.