MONASH University

Workload-Aware Data Management in Shared-Nothing Distributed OLTP Databases

Joarder Mohammad Mustafa Kamal

BScEngg (*Hons*), MRes

A thesis submitted for the degree of *Doctor of Philosophy* at
Monash University in 2016
Faculty of Information Technology

Main Supervisor: Professor Manzur Murshed
Associate Supervisor: Associate Professor Joarder Kamruzzaman
External Supervisor: Professor Rajkumar Buyya
External Supervisor: Professor Mohamed Gaber

Copyright Notice

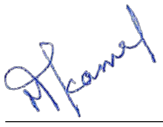
Abstract

More than 3.26 billion Internet users are using Web and mobile applications every day for retail services and businesses processing; information management and retrievals; media and social interactions; gaming and entertainments, etc. With the rapid growth in simultaneous users and data volume, such applications often rely on distributed databases to handle large magnitudes of On-Line Transaction Processing (OLTP) requests at the Internet scale. These databases employ user-defined partitioning schemes during their initial application design phase to scale-out concurrent writing of user data into a number of shared-nothing servers. However, processing Distributed Transactions (DTs), that involve data tuples from multiple servers, can severely degrade the performance of these databases. In addition, transactional patterns are often influenced by the frequent data tuples in the database, resulting in significant changes in the workload that none of the static partitioning schemes can handle in real-time. This thesis addresses the abovementioned challenge through workload-aware incremental data repartitioning. The primary goal is to minimally redistribute data tuples within the system with a view to minimising the adverse impact of DT without adversely affecting the data distribution load balance. Furthermore, to support both *range* and *consistent-hash* based data partitions used in OLTP systems, a distributed yet scalable data lookup process, inspired by the roaming protocol in mobile networks, is introduced. Moreover, additional challenge lies on how to reduce the computational complexities while processing large volume of transactional workload very quickly to make real-time decisions. These complexities are addressed in three distinct ways - hide the workload network complexities during analysis by using different level of abstractions; find the appropriate size of observation window to decide how far to look back in the past for workload analysis; and finally, incrementally (i.e., how frequently) trigger the repartitioning process based on a threshold. Some well-defined Key Performance Indicator (KPI) metrics for database repartitioning are defined and rigorous sensitivity analyses are performed on different types of transaction generation models and observation window sizes. In the first stage, graph-theoretic higher level abstractions such as graphs, hypergraphs, and their compressed forms are used to deliver sub-optimal repartitioning decisions using graph *min-cut* techniques. In order to deliver optimal de-

cisions, a greedy approach is further developed, which ranks the transactions in terms of the measure of repartitioning objectivity and performs transaction-level optimisation to maximise a particular KPI. Finally, transactional stream mining is used to construct a representative sub-network of frequent transactions and perform transaction-level adaptive repartitioning, which eventually reduces the computational complexity, while concomitantly improving the performance due to filtering out a significant volume of infrequent transactions, that have little bearing on the KPIs. Simulation-based experimental evaluations demonstrate that the proposed workload-aware incremental repartitioning schemes outperform the existing static and dynamic repartitioning approaches to be considered effective for production deployment.

Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature: 

Print Name: Joarder Mohammad Mustafa Kamal

Date: 21 February 2017

Acknowledgements

First and foremost, I would like to express my gratefulness to the Almighty, Allah, the most generous for blessing me with the physical and mental ability, sustenance, and patience to undertake this research successfully. Next, I need to thank my parents and parent-in-laws for continuously encouraging me to purse my dream. Without their ultimate sacrifice, it would not be possible for me to reach this end.

I am extremely fortunate to have Prof. Manzur Murshed as my main supervisor. Without his persistent guidance, insightful directions, and support, this research would not have been successfully completed. I am also indebted to him and his family for allowing me the time over a number of weekends to have research meetings at their residence when I moved to Sydney for work. I am also thankful to A/Prof. Joarder Kamruzzaman, Prof. Rajkumar Buyya, and Prof. Medhat Gaber for their valuable advices and guidelines. Additionally, I am grateful to Dr. Gour Karmakar, A/Prof. Madhu Chetty, Prof. Sue McKemmish, and Dr. Mortuza Ali for often enlightening my ideas and findings. I am also thankful to Prof. Donald Kossmann, Dr. Carlo Curino, Dr. Evan Jones, and Dr. Djellel Difallah for their consecutive correspondences, suggestions and comments on my works. I specially thank the Australian Government and Monash University for supporting my research with the IPRS and APA scholarships. I also thank the staffs at the Monash Campus Cluster (MCC), the Faculty and Gippsland School of IT, and Monash University Institute of Graduate Research (MIGR) for all of their administrative supports towards the successful completion of my research. A special thanks to Mrs. Freda Webb for her exceptional cooperation as my student advisor. I would also like to thank Dr. Gillian Fulcher for proofreading this thesis.

I am ever thankful to my dear friends and well-wishers Fasahat Siddiqui, Ahammed Shereif, Ahsan Raja Chowdhury, Rumana Nazmul, Hasanul Ferdaus, Ali Naqi, Tori Stratford, Mamun Abdullah, Shahriar Kaisar, and Shakif Azam for their selfless support, encouragements, and motivations that never let me feel alone. Finally, I specially thank my wife, Rushmila, to share this enormous journey with me and sacrifice the most in doing so. At the beginning of my PhD, she gave up her career to join me in Australia and has shared all the difficult moments in all these past years. Without her everlasting love and continual support, I would not be able to submit my thesis today.

Dedicated to my parents, and to my wife who sacrificed the most for this research.

Abbreviations

2PC	Two-Phase Commit
2PL	Two-Phase Locking
3PC	Three-Phase Commit
ACID	Atomicity, Consistency, Isolation, and Durability
ADMS	Adaptive Data Management Systems
AR	Application Requester
ARHC	Association Rule Hypergraph Clustering
AS	Application Server
BASE	Basically Available Soft-State Eventual Consistency
BC	Base Classification
CGR	Compressed Graph Representation
CHR	Compressed Hypergraph Representation
DB	Database
DS	Database Server
DBMS	Database Management Systems
DDB	Distributed Database
DDBE	Distributed Database Environment
DDBMS	Distributed Database Management Systems
DFCI	Distributed Frequent Closed Itemset
DSM	Data Stream Mining
DT	Distributed Transaction
FCI	Frequent Closed Itemset
FCT	Frequent Closed Tupleset
FI	Frequent Itemset

Abbreviations (continued)

FRTC	Full Representative Transaction Classification
FT	Frequent Tupleset
GR	Graph Representation
GSM	Global System for Mobile Communications
HR	Hypergraph Representation
HRP	Hourly-based Repartitioning
KPI	Key Performance Indicator
MCM	Maximum Column Matrix
MNDT	Moveable Non-distributed Transaction
MSM	Maximum Sub-Matrix
MST	Minimum Support Threshold
MVC	Model View Controller
NAS	Network Attached Storage
NC	No Classification
NDFCI	Non-distributed Frequent Closed Itemset
NDT	Non-distributed Transaction
NIPDM	Net Improvement Per Data Migration
NMNDT	Non-moveable Non-distributed Transaction
NoSQL	Not Only SQL
NRP	No Repartitioning
OLAP	Online Analytic Processing
OLTP	Online Transaction Processing
PACELC	If **P**artition then **A**vailability and **C**onsistency **E**lse **L**atency and **C**onsistency
PRTC	Partial Representative Transactional Classification
RDBMS	Relational Database Management Systems
REST	Representational State Transfer

Abbreviations (continued)

RM	Resource Manager
SAN	Storage Area Network
SD	Shared-Disk
Semi-FCI	Semi-Frequent Closed Itemset
SN	Shared-Nothing
SRP	Static Repartitioning
SSJ	Stochastic Simulation in Java
SQL	Standard Query Language
TCO	Total Cost of Ownership
TM	Transaction Manager
TPC	Transaction Processing Performance Council
TRP	Threshold-based Repartitioning
USL	Universal Scalability Law
WMNIPDM	Weighted Mean Net Improvement Per Data Migration
WS	Web Server

Notations

α	Exponential averaging coefficient for transactional recurrence period
A	The set of incident transactions
A_c	Associativity score of a transaction
$\mathcal{A}$	The set of adjacent tuples
β	Load-balance factor for graph *min-cut* clustering
γ	Span reduction factor
C_l	Compression level
C_v	Coefficient of variance
δ	A data tuple
$\bar{\delta}$	A tupleset
Δ	The tuple set of a distributed transaction
$\mathcal{D}$	Data volume
D_m	Inter-server data migrations
e_g	An edge
e'_g	A compressed edge
E_g	The set of edges
e_h	A hyperedge
e'_h	A compressed hyperedge
E_h	The set of hyperedges
ε	Maximum allowed imbalance ratio
θ	Exponential averaging weight
$\mathcal{F}$	The set of semi-FCI
$\mathcal{G}$	A graph
$\mathcal{G}_c$	A compressed graph
$\mathcal{H}$	A hypergraph
$\widetilde{\mathcal{H}}$	A hypergraph of tuplesets

Notations (continued)

$\mathcal{H}_c$	A compressed hypergraph
I_d	Impact of distributed transactions
κ	Weighted mean net improvement per data migration
λ	Exponential averaging coefficient for greedy heuristic based repartitioning
Λ	The set of semi-frequent closed tuplesets
L_b	Server-level load balance
μ	Mean
m	A migration plan
M_g	The total number of inter-server data migrations
$\mathcal{M}$	The potential list of data migration plans
p	The probability that a new transaction will appear in an observation window
$\mathcal{P}$	The set of logical database partitions
$\wp$	Power set
η	Average number of new transactions in an observation window
r	Relaxation rate
Υ	Observed period of transactional recurrence
$\tilde{\Upsilon}$	Expected period of recurrence
$\mathcal{R}$	Transaction generation rate
R_t	Repartitioning triggering threshold
σ	Standard deviation
∂	Minimum support threshold
S	The set of shared-nothing physical database servers
ζ	k-way balanced clustering solution
τ	A transaction
τ_d	A distributed transaction
$\tau_{\hat{d}}$	A non-distributed transaction

Notations (continued)

t	System time
T	The set of transactions
T_d	The set of distributed transactions
$T_{\hat{d}}$	The set of non-distributed transactions
$T^{\bowtie}$	The set of distributed transactions incident to another transaction
$\mathcal{T}$	The set of transactions containing any particular tupleset
U	The target proportion of unique transactions in an observation window
V	The set of vertices
V'	The set of compressed vertices
$\widetilde{V}$	The set of association rule hypergraph clusters
w	Weight value
ψ	The residing server of a data tuple
W	Transaction-sensitive sliding window
$\mathcal{W}$	Workload observation window
$\mathcal{X}$	Incremental repartitioning solution

Publications

1. **J. Kamal** and M. Murshed, Distributed database management systems: Architectural design choices for the cloud." *Cloud Computing Challenges, Limitations, and R&D Solutions, Computer Communications and Networks*, Z. Mahmood, Ed. Springer International Publishing, pp. 23-50, 2014.

2. **J. Kamal**, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable OLTP applications." *Future Generation Computer Systems*, vol. 56, no. C, pp. 421-435, Mar. 2016. [**IF: 2.430**]

3. **J. Kamal**, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable cloud applications." *In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*. London, UK, pp. 213-222, Dec. 2014.

4. **J. Kamal**, M. Murshed, and M. Gaber, "Progressive data stream mining and transaction classification for workload-aware incremental database repartitioning." *In Proceedings of the 2014 IEEE/ACM International Symposium on Big Data Computing, BDC '14*. London, UK, pp. 8-15, Dec. 2014.

5. **J. Kamal**, M. Murshed, and M. Gaber, "Predicting hot-spots in distributed Cloud databases using association rule mining." *In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*. London, UK, pp. 800-806, Dec. 2014.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

"Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;"
– Robert Frost, *The Road Not Taken (1916)*

With the widespread availability of home and mobile Internet services even to the remote places on the earth, man–machine interactions have increased to a level never seen before. Internet facilities are nowadays freely available at most shopping centres, coffee shops, restaurants, airports, train stations, and bus stops, offices and public places as a guest, etc. At the same time, low-cost smartphones, tablets, notebooks, and laptops that have computing capabilities and resources matching those of a desktop machine are readily available and the number of mobile application users is increasing rapidly. Ubiquitous access to the Web enables us to use real-time applications for navigating information on the move, such as real-time timetables for public transportation; mobile banking transactions; accessing e-books, news, media, and audio/video streaming services; gaming on-line with multi-players; sharing documents, photos, and videos; and, finally, interacting nonstop within the on-line social networks. It has been estimated that there will be 1.5 mobile devices per capita (11.1 billion devices globally) by the end of 2020, transferring an estimated monthly rate of 30.6 exabytes of data traffics [1]. The enormous growth of the World Wide Web within a very short time creates the need to develop a new class of interactive, real-time, and mobile-friendly Web applications for the end users.

Figure 1.1 presents a very high-level overview of a modern Web application following a request-reply cycle. There are three main components: 1) the application's

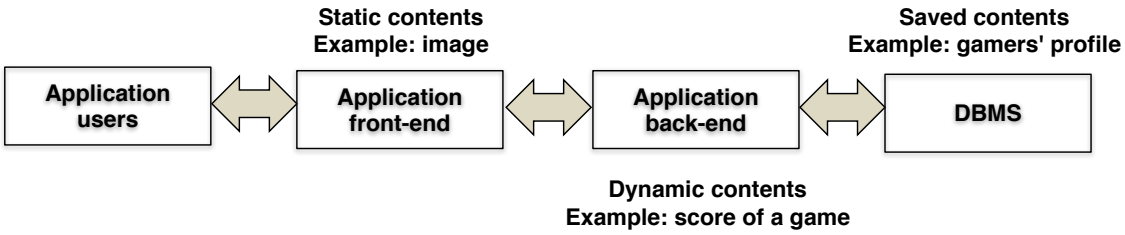


Figure 1.1: The overview of a modern Web application with request-reply flows

front-end, 2) the application's back-end, and 3) data storage. The challenge of building a new class of applications for Web users comes with a set of requirements to be guaranteed at the application and data service provider's end. From the application developer's perspective, this new class of Web applications should be built around instantaneous request-reply cycles using asynchronous communication media without end-users noticing any delays due to back-end computing and data processing. From a data management perspective, this aspiration is rather difficult to meet as the application data needs to be stored in ways that guarantee consistency and durability, concurrent access by the application code, and being available all the time for reliable retrievals. Finally, the overall infrastructure supporting modern Web applications needs to be highly scalable in order to meet rapid user growth and service demands. Legacy systems for on-line Web data management were primarily developed to support sophisticated financial and high-end enterprise applications, and in this modern era, they are not fully capable of serving this new class of Web traffic workloads.

Today, the largest proportion of real-time Web traffic is driven by On-line Transaction Processing (OLTP) applications comprising short-lived transactions,[1] typically accessing less than a hundred data tuples[2] at any moment and requiring high responsiveness from the application. Examples of such workloads are typical Web interactions and on-line applications; e-commerce, order processing, and financial transactions; multimedia, gaming, and sharing; social networking, etc. OLTP workloads typically require a *database schema* and an exposed *entity-relationship* data model. Naturally, Relational Database Management Systems (RDBMS) are the first-class citizens for such

[1] Transaction is a unit of work performed by a relational database management system.

[2] A table row in a relational database is called a *tuple* in a table relationship and is the smallest unit of a logical database.

Web-oriented workloads and have dominated the Internet world over the past three decades. With the rapid growth in data-oriented computing domain, real-time Internet applications deployed in the Web need to scale-out in an unparalleled way to support millions of simultaneously connected users. Nevertheless, without scaling out the data management layer it's not possible to scale-out the back-end application and front-end presentation layers of any classic 3-tier Web application as shown in Figure 1.1.

Scaling out real-time Web applications often relies on shared-nothing (SN)[3] distributed databases for workload adaptability achieved through horizontal data partitioning. In this way the underlying database layer can increase parallel transaction processing, balance workload and data distribution, and eventually the overall system throughput. However, Distributed Transactions (DTs) that access data tuples from multiple physical machines containing a portion of the required data need to lock the accessing data tuples globally in order to perform consistent and durable transactional executions. Unfortunately, the process of distributed transaction processing adversely impacts the database scalability while executing *distributed transactional commit* and *concurrency control* protocols to maintain the strong Atomicity, Consistency, Isolation, and Durability (ACID) guarantees in an RDBMS. Therefore, minimising the impact and proportion of DTs in the workloads is the primary challenge for interactive and real-time Web applications serving concurrent application data requests from end-users.

1.1 Motivations and Research Questions

In a shared-nothing Distributed Database Management System (DDBMS) cluster, individual database tables are either range- or hash-partitioned at the preliminary design step and logical data chunks are placed accordingly within each Database Server (DS) node. However, such static partitioning decisions do not work well in the presence of dynamic OLTP workloads and often require significant administrative interventions to manually *repartition* the logical database. This process involves migrating data tuples physically over the network to another DS node causing potential system

[3] A shared-nothing (SN) architecture is a distributed computing architecture where each participating node is independent and does not share computing and storage resources with any another.

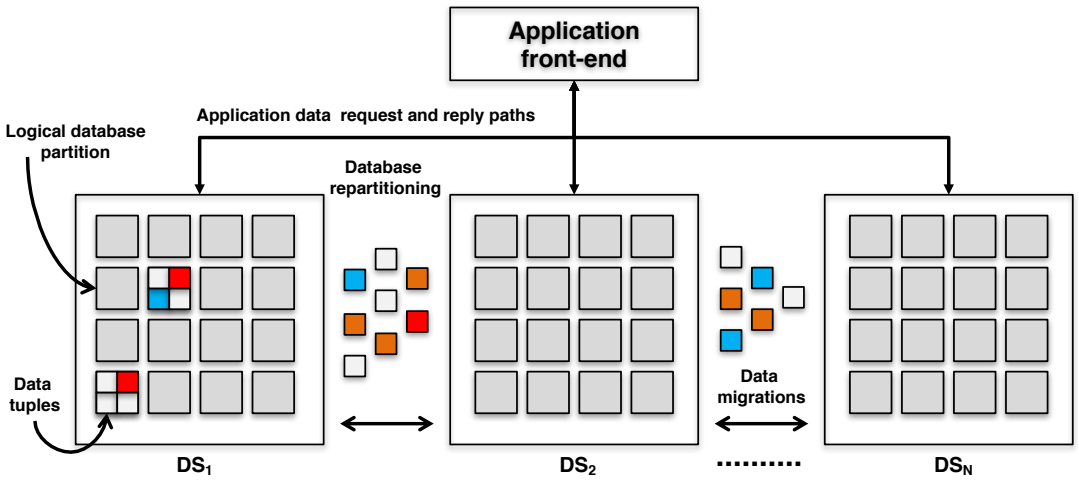


Figure 1.2: High-level overview of a repartitioning process within a shared-nothing DDBMS

downtimes and application inaccessibility for the end-users. Figure 1.2 shows a high-level overview of a database repartitioning process. Further to this challenge, real-time OLTP applications deployed in the Web often need to scale-up and scale-down elastically depending on the change in Web traffic, workload size, and data popularity. In such situations, instantaneous database repartitioning and on-line live data migration are necessary to guarantee acceptable system throughput and overall responsiveness. Afterwards, as the crisis situation eliminates when the sudden rise of workload spikes smooth down, the current database partitioning layout is deemed useless and requires another repartitioning.

The database administrator needs to define an initial data partitioning strategy – selecting a partitioning column for hashing, a list of column values for selection, or range keys on the database tables. A centralised data lookup table needs to be maintained for *range-partitioning* and periodical partition rebalancing. Alternatively, in hash partitioning, rehashing of keys is required while adding an additional DS or removing any faulty ones from the cluster. All these operations involve massive inter-server physical data migrations and in many cases potential downtimes. Fortunately, consistent hashing [2] based data partitioning solves these problem by using a fixed number of equal sized logical partitions and providing consistent hash keys for the individual data tuples distributed within a circular key ring for scalable lookup operations. However, implementing a database tier based on consistent hashing is not trivial and is better

suited to a key-value type data model. Legacy RDBMSs, on the other hand, can adopt a composite partitioning scheme that is either a combination of range-list or range-hash based partitioning. Nevertheless, inappropriate database schema design and partitioning may create potential data hotspots within the cluster and imbalances in workload access and data volume distributions. Eventually such situations lead to resource contentions, performance degradations, disk failures, and downtimes.

To support rapid increase in data volume and elastic growth of the DDBMS cluster, a scalable OLTP system should be able to scale-out and scale-in seamlessly by adding or removing new DS nodes. This includes added challenge to the existing data partitioning layout for rebalancing existing partitions within physical cluster nodes. Existing tools used by large-scale OLTP service providers like *Vitess* [3] from *YouTube* or *JetPantsJetPants2015* from *Tumblr* support automatic partition management and load balancing. However, without continually monitoring, detecting, and adapting workload dynamics it is administratively expansive, in terms of operational management, to guarantee the elastic DDBMS scalability of modern OLTP applications. In the 2014 Beckman Report on Database Research [4] published by ACM, seven primary challenges are mentioned. They are 1) elasticity for data repartitioning and migrations, 2) replication across geo-distributed data centres, 3) system administration and tuning, 4) multi-tenancy, 5) SLA and QoS for managed DDBMS services, 6) data sharing, and 7) data management between public and private computing domains. In a nutshell, the primary research questions to be answered in this research are how to 1) repartition the database independently of the number of physical servers; 2) incrementally repartition to reduce the impacts of DTs which arise from workload characteristics, 3) perform on-the-fly data migration and minimise its cost, 4) perform scalable data lookup in a distributed manner, and 5) maintain data distribution and workload execution load-balance on the physical server-level and logical database partition-level.

1.2 Significance

Since 2006, with the emergence of Cloud computing providing managed infrastructures, as well as structured and unstructured database services as a pay-as-you-go

model, small-to-medium scale Web applications have started migrating from legacy platforms to the Cloud. In recent years, this trend has gained such popularity that large-scale OLTP application service providers also modularise their architecture to move under a managed service operation. This includes existing managed database platforms for high scalability rise from the development of Amazons DynamoDB [5], Google's BigTable [6], and Facebook's Cassandra [7] since early 2007. Since then, there has been a clear trend in abandoning relational data models and SQL in favour of adopting simpler key-value data models and high scalability comprising ACID guarantees. Over the years, by trading-off between *Consistency*, *Availability*, and network *Partition* tolerance following the CAP principle [8], a significant number of NoSQL systems have been evolved and adopted by many without understanding their proper use cases. These NoSQL systems adopt a new paradigm of database guarantees named *BASE* [9].[4] The *E* in *BASE*, which stands for *Eventual Consistency* [10, 11], guarantees that, in the face of inconsistency, the underlying system should work in the background to catch up with the consistent states. The idea is that, in many cases, it is hard to distinguish the inconsistent state of a data tuple from the end-user perspective that is bounded by different staleness criteria like time-, value-, or update-bounded staleness. Without exploring the design space and use cases for OLTP systems in the Web, many small- to medium-scale OLTP applications by adopting a NoSQL data management layer as an inner-tier database back-end, greatly suffer from trading-off *consistency* for *network partition* tolerance.

The architectural choices for different service tiers in an OLTP system vary, based on the application requirements and business goals. If an application requires serving highly consistent and durable data to the end-users every time they initiate service requests, then high scalability will be hard to achieve and high availability will not be visible due to low responsiveness. Otherwise, if an application can successfully satisfy the end-users with an acceptable level of inconsistency, then both high scalability and availability is achievable. In reality, the probability of network partitions in today's highly reliable data centres is quite rare although short-lived partitions are common in wide-area networks. According to the CAP principle, DDBMSs should be both *available* and *consistent* when there are no network *partitions*. Still, due to high volume

[4] BASE stands for Basically Available Soft-State Eventual Consistent

user requests or sudden disk failure, the responsiveness of inner-tier RDBMS services may lag behind. In such cases, timely responses of the submitted requests are clearly preferable. Thereby reducing *latency* by using cached data and remaining *available* to the end-users. The ultimate goal is to build a scalable OLTP application that remains *available* and *responsive* to the end-users even at the cost of tolerable inconsistency which can be deliberately engineered in the application logics.

From these recent understandings and revisiting the architectural design space, the appropriate strategies for operating an RDBMS is now understood in the context of the Cloud paradigm. Although the simplicity and manageability of relational data models never lost their attractions in the wider community these years, it's now better understood in the context of large-scale system development striving for high-scalability. Since 2010, the *NewSQL* movement started architecting high scalable RDBMSs deployed in shared-nothing clusters without sacrificing strong transactional and consistency requirements. As a large number of medium- to large-scale enterprises and general-purpose OLTP service providers need to process billions of transactional queries simultaneously, the challenge to perform dynamic and transparent database repartitioning becomes a top priority task to accomplish scalable workload-aware data management in the distributed RDBMS domain. The database community is still trying to pull different things together to make the legacy shared-nothing RDBMS clusters more adaptable for the elastic Cloud platforms, and search is not over yet. In a 2013 talk [12], database pioneer MIT Professor Michele Stonebraker admits 'For OLTP, the race for the best data storage designs has not yet been decided, but there is a clear indication of classic models being plain wrong'.

1.3 Philosophical Approaches

In this dissertation, we aim to solve the above-mentioned workload-aware incremental database repartitioning problem that exists in the OLTP domain. The challenge of performing such repartitioning can be addressed from different standpoints. One of the interesting approaches already adopted in many existing literature is to use some mathematical abstractions to hide the underlying complexities in the transactional workload and operate at a birds eye view level to find repartitioning solutions.

In order to reduce the complex OLTP dynamics to a set of essential characteristics, transactions can be simply represented by a network of nodes representing the transactional data tuples from the database queries and adding links to connect them together to represent individual transactions. This simple mathematical representation allows us to use well-established graph theoretic approaches to sub-optimally group these network nodes into a set of buckets so that most of the transactional operations can retrieve all of its required data tuples from a single physical server in a database cluster. This eventually converts many of the DTs in the workload to non-DTs (NDTs), which in turn reduces the adverse impacts of DTs in the OLTP transaction processing layer. However, once the network size tends to grow rapidly with the number of increasing transactions in a system, this theoretically stronger approach can suffer immensely in terms of optimal repartitioning decision making.

Another approach to deal with this challenge is to remove the abstraction layer and directly operate at the transactional-level. However, finding the best possible way to convert a DT to a NDT becomes even more challenging as there can be many possible data migration plans to consider. A repartitioning scheme needs to go through all of these plans exhaustively to find the best data movement plan so that some of the transactional tuples can be redistributed within the physical servers. A well-known way to do so is to use some greedy optimisation heuristics where the best solution at individual stages are expected to lead to near-optimal global solution. We could use this simple strategy for transactional repartitioning as well. By optimally finding the best data redistribution plan for converting individual DTs to NDTs, it is possible to reach close to global optima, which can significantly reduce the system wide impacts of DTs. The primary challenge in this approach is that the local search space for optimal decision can grow so large that globally finding even a near-optimal result may become infeasible. As a remedy, any special heuristic might be applied to reduce the size of the local search space in order to quickly obtain the repartitioning decisions.

An interesting way to handle the abovementioned computational complexity challenges would be to only work with a selective set of transactions that dominatingly represent the entire OLTP workload. By exploiting the heavily lop-sided distribution of the highly-frequent transactions that constitute the selective set, the size of the problem can be reduced significantly to a level where real-time repartitioning decision may

be possible on-the-fly even for substantially large-scale distributed OLTP systems. With the recent advancements in data mining research, we can utilise on-line data stream mining techniques to find the frequently appeared data tuplesets in the transactional workload. Later, these tuplesets can be used to identify the representative set of transactions in the system that dominates the entire transactional processing. This important data mining knowledge can be used both globally for all transactions or locally for individual ones to find the best data redistribution plan to reduce the overall number of DTs in the system gradually over time. Considering practical adaptability, the later approach would have been more appropriate to handle the dynamic nature of the OLTP workloads.

Overall, we can observe interesting conflicts among these three different approaches from a philosophical standpoint. At one end, it is possible to find a sub-optimal repartitioning solution for reducing the adverse impacts of DTs by hiding transactional complexities using an abstraction layer. On the other hand, the greedy approach can provide a near-optimal decision but the search space to find that solution can potentially be too large to render it infeasible for any practical use. Finally, we have the transactional mining approach which relies on generating an approximate solution but at the same time provide the flexibility to run the repartitioning process dynamically and as frequently as the system may require. In this thesis, we aim to study and investigate all of these three approaches to understand their strengths, shortcomings, and implications from both theoretical constructions and experimental observations. Additionally, to compare the adaptability and performance of these repartitioning schemes for practical use, we may also need to establish appropriate KPIs with proper definitions by understanding their corresponding physical interpretations.

1.4 Research Challenges

The OLTP characteristics are changing the landscape for how distributed RDBMSs are used to support traditional Web applications. Due to the rapid increase in OLTP applications and services in the recent years to serve real-time transactional queries, it is essential to partition the database over multiple physical servers. However, the

real challenge resides in how we can effectively repartition this ever growing database over time in accordance with the change in workload behaviours discussed earlier. Furthermore, any scalable incremental database repartitioning scheme should be able to provide a set of key performance indicators (KPIs) to primarily evaluate how it works in terms of reducing DTs and their overall effect on the system along with how this might affect the data distribution load-balance within the physical machines, and finally how much data needs to moved physically. An obvious challenge is to find the right balance between reducing the adverse impacts of DTs and load-balance by performing a reasonable amount of physical data migrations while not exhausting any system resources. Overall, the computational complexities of such a scheme trading off multiple KPIs should be done at a minimum to get an instant repartitioning decision for automated database administration during high surges of transactional load. In the context of the abovementioned aim and object, the relevant key philosophical challenges are identified and studied in this thesis. They are summarised below.

(i) Understanding of the physical interpretations of the consequences of applying any incremental repartitioning scheme, namely a) the impact on the RDBMS performance for processing significant numbers of DTs generated by the OLTP workloads, b) the impact on the underlying storage systems arising from data distribution imbalance, and c) the impacts of physical data migrations over the network and its relationship with the impact on DT processing and server-level load-balance.

(ii) Development of the KPIs that relate the physical consequences of incremental repartitioning which are simple enough to manoeuvre by the system owners and provide operational intelligence to rapidly deal with critical situations dynamically.

(iii) Investigation of graph theoretic and greedy heuristic based incremental repartitioning independent of the physical properties of an RDBMS deployment, which will allow the development of scalable solutions and an understanding of their worst-case scenarios for different workload characteristics.

Figure 1.3 presents the potential challenges in database systems research based on

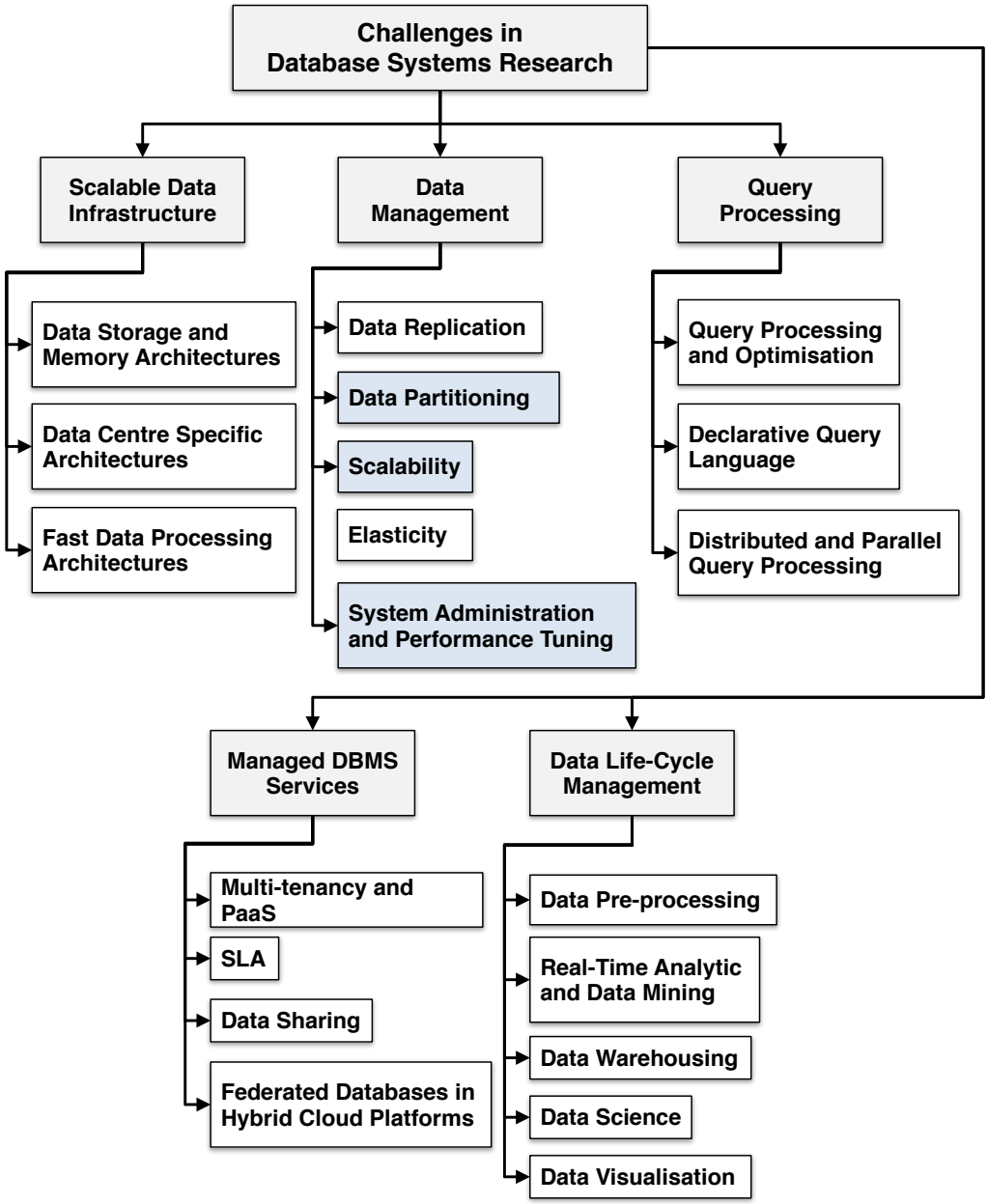


Figure 1.3: The potential challenges and targeted contributions in database systems research

the 2014 Beckman Report on Database Research [4]. In light of the above research challenges, this thesis targets the specific areas in the light blue coloured rectangles.

1.5 Aim and Objectives

The aim of this thesis is to contribute workload-aware incremental data repartitioning techniques for shared-nothing distributed RDBMSs for supporting scalable and elastic Web applications driving OLTP workloads. To fulfil this aim, this thesis thus focuses on the following key objectives in developing novel incremental data repartitioning models:

(i) To carry out detailed literature review in order to identify the shortcomings of existing data partitioning strategies for RDBMSs, the impacting factors that reduce scalable transactional processing, and to provide a comprehensive and structured understanding of the problem domain.

(ii) To develop repartitioning quality metrics to evaluate how well different incremental repartitioning schemes perform to decrease the impacts of distributed transactions rapidly while maintaining data distribution balance by performing minimum physical data migrations.

(iii) To develop a simulation model for transactional workload generation independent of database engines and transactional processing in order to evaluate different incremental repartitioning schemes in a cohesive manner.

(iv) To develop a scalable and unified data lookup method to work with dynamic repartitioning schemes supporting both *range-* and *consistent-hash partitioned* RDBMSs.

(v) To develop demand-driven incremental repartitioning schemes with diverse workload characteristics targeting three primary optimisation goals: 1) minimising the impacts of DTs, 2) maintaining server-level data distribution load-balance, and 3) minimising inter-server physical data migrations.

(vi) To devise transaction classification and data stream mining techniques to analyse the minimum number of transactions in a workload observation window in order to make adaptive repartitioning decisions.

(vii) To validate and verify the performance and adaptability of different incremental repartitioning schemes under different OLTP workloads.

1.6 Thesis Contributions

Both qualitative and quantitative research methodologies are used to meet the above intellectual challenges in this thesis. While qualitative measures are used in understanding the physical properties of the problem domain and their associations with the underlying systems, quantitative methods are applied to turn those rational measures into practical solutions. The key contributions of the thesis are summarised below.

(i) A scalable distributed data lookup technique supporting both *range-* and *consistent-hash partitioned* databases

(ii) A novel transaction generation model which supports controlled transactional recurrence within a given workload observation window for a fixed proportion of unique transaction generations

(iii) A set of repartitioning KPIs with physical interpretations to measure the performance and quality of incremental repartitioning of distributed OLTP databases

(iv) A novel transaction classification method which classifies DTs and *moveable* non-distributed transactions (MNDTs) for graph theoretic incremental repartitioning

(v) A detailed analysis of different workload network representations using graph, hypergraph, and their compressed forms in graph *min-cut* clustering based incremental data repartitioning for static, hourly, and threshold defined scenarios

(vi) Two innovative cluster-to-partition mapping algorithms for graph clustering based repartitioning that favours the current partition of the majority tuples in a cluster. The *many-to-one* version minimises data migrations alone and the *one-to-one* version reduces data migrations without affecting load-balance

(vii) A greedy heuristic approach to incremental repartitioning that can near-optimally find repartitioning solutions that achieve the highest wighted net gain of reducing the impacts of DTs and maintaining load-balance per physical data migration

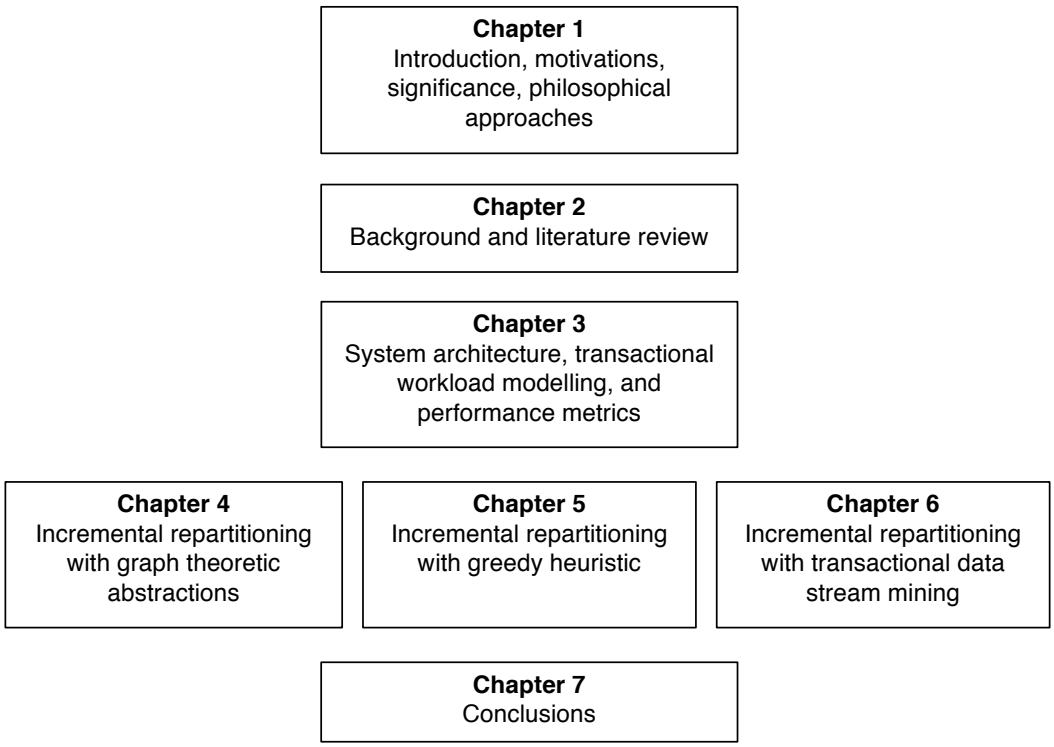


Figure 1.4: An overview of thesis organisation and contributions

(viii) A set of extended transaction classification methods based on transactional data stream mining to identify the partial and full representative set of transactions in the workload network

(ix) An association rule hypergraph clustering (ARHC) based scheme which utilises the transactional data stream mining technique to find the transactional associativity and based on this measure perform atomic incremental repartitioning to reduce the impacts of DTs

1.7 Thesis Organisation

The rest of chapters in the thesis are organised by fulfilling the research aim and objectives, while providing details of the contributions mentioned above. Figure 1.4 presents a conceptual overview of the thesis organisation and core contributions in each chapter. The rest of the thesis is organised as follows:

(i) Chapter 2 presents a detailed literature review on distributed data management for RDBMSs and existing approaches for data repartitioning along with their shortcomings. This chapter is partially derived from:

- **J. Kamal** and M. Murshed, Distributed database management systems: Architectural design choices for the cloud." *Cloud Computing Challenges, Limitations, and R&D Solutions, Computer Communications and Networks*, Z. Mahmood, Ed. Springer International Publishing, pp. 23-50, 2014.

(ii) Chapter 3 provides the background and preliminaries necessary for the workload-aware incremental data repartitioning schemes to be presented in the following chapters. The scalable distributed data lookup technique is presented which is one of the essential components to support incremental repartitioning in both *range-* and *consistent-hash partitioned* distributed RDBMSs. The construction of the proposed transaction generation model is discussed in detail with confirmatory results. Three repartitioning KPIs are also presented with relating physical interpretations. This chapter is partially derived from:

- **J. Kamal**, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable OLTP applications." *Future Generation Computer Systems*, vol. 56, no. C, pp. 421-435, Mar. 2016.
- **J. Kamal**, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable cloud applications." *In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*, London, UK, pp. 213-222, Dec. 2014.

(iii) Chapter 4 includes the detail of graph *min-cut* clustering based workload-aware incremental repartitioning schemes where workload networks are represented as graph, hypergraph, and their compressed forms. It also includes the novel transaction classification technique and two unique cluster-to-partition mapping mechanisms for maintaining load-balance and minimising physical data migrations. A detailed analysis is carried out for both *range-* and *consistent-hash partitioned* setups comparing static and incremental repartitioning approaches. This chapter is partially derived from:

- **J. Kamal**, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable OLTP applications." *Future Generation Computer Systems*, vol. 56, no. C, pp. 421-435, Mar. 2016.
- **J. Kamal**, M. Murshed, and R. Buyya, "Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable cloud applications." *In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*. London, UK, pp. 213-222, Dec. 2014.

(iv) Chapter 5 introduces the greedy heuristic based approach to incrementally repartition the underlying database to find a near-optimal repartitioning decision. The detail analysis unravels the relationships between the three KPIs of the impacts of DT, load-balance, and data migrations.

(v) Chapter 6 extends the transaction classification technique described in Chapter 4 by exploiting transactional data stream mining to identify the frequently accessed tuplesets and to construct a workload network containing the partial and full set of representative transactions for graph theoretic incremental repartitioning. This chapter further presents an atomic incremental repartitioning scheme using association rule based hypergraph clustering (ARHC). Experimental results are presented to show the effectiveness of the approximate repartitioning decisions that can be adopted in practice. This chapter is partially derived from:

- **J. Kamal**, M. Murshed, and M. Gaber, "Progressive data stream mining and transaction classification for workload-aware incremental database repartitioning." *In Proceedings of the 2014 IEEE/ACM International Symposium on Big Data Computing, BDC '14*. London, UK, pp. 8-15, Dec. 2014.
- **J. Kamal**, M. Murshed, and M. Gaber, "Predicting hot-spots in distributed Cloud databases using association rule mining." *In Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC '14*. London, UK, pp. 800-806, Dec. 2014.

(vi) Chapter 7 concludes the thesis by summarising the contributions and key findings as well as providing future directions for research.

Chapter 2

Background and Literature Review

The previous chapter introduced the motivation, significance, and challenges of the research goal to develop workload-aware incremental repartitioning techniques for shared-nothing distributed OLTP databases to support scalable Web applications. In this chapter, the preliminaries of distributed RDBMS and associated backgrounds of modern Web applications are discussed in detail, as are accompanied by the relevant advancements in academic research and in industry.

2.1 Introduction

The challenges of managing consistency and concurrency of a transactional Web application often relies on the underlying architecture of its RDBMS. Meanwhile, to achieve high resistance to faults and high scalability without compromising operational performance, the RDBMS deployment often requires it to *replicate* and *partition* all or some of the database tables with the overheads of managing distributed transactional commits and concurrency control. The distribution of data within a cluster of physical machines is often implemented by adopting a shared-disk (*SD*) or a shared-nothing (*SN*) architecture. However, due to the dynamic nature of OLTP workloads, the underlying DDBMS needs to regulate the initial data replication and partitioning schemes administratively (manually) or automatically over time. In this chapter, the elemental properties, architectures, components, and strategies to manage distributed OLTP databases are discussed in light of the existing literature.

The rest of the chapter is organised as follows: Section 2.2 discusses the transactional properties of a distributed OLTP system. The deployment and data management architectures for shared-nothing OLTP DDBMSs are presented in detail at Section 2.3. Section 2.4 describes the challenges for scaling out multi-tier OLTP applications in a

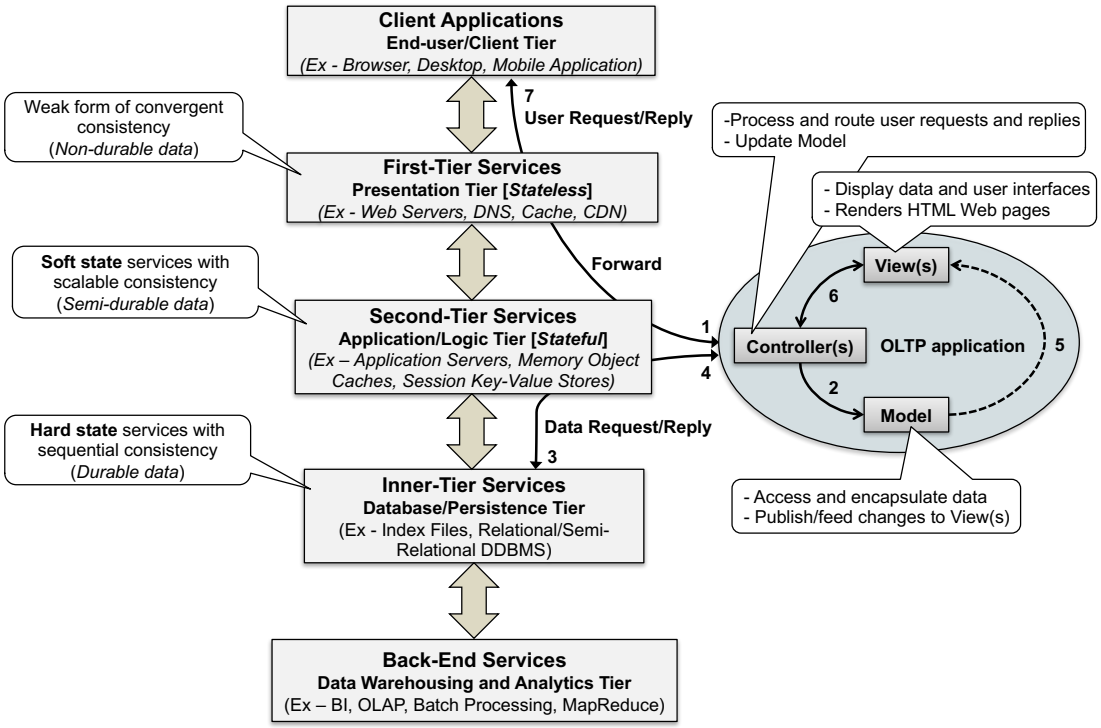


Figure 2.1: The high-level overview of an OLTP system deployed in the Web with corresponding control and data flows (from **1** to **7**) following an MVC software architectural pattern

distributed environment, followed by the challenges of scaling out distributed OLTP databases in Section 2.5. In Section 2.6, the necessary preliminaries for understanding existing workload-aware data management approaches are presented. Section 2.7 presents a detailed discussion on the contributions and shortcomings of the existing repartitioning schemes in the literature, followed by a conceptual framework covering specific aspects implementation to compare with the existing work. Finally, Section 2.8 summarises and concludes the chapter.

2.2 On-line Transaction Processing Systems

OLTP is a class of interactive and real-time information management systems that support transaction-oriented applications typically deployed in the Web for individuals and enterprises. Transaction processing involves financial transactions and e-commerce; telecommunication; booking and reservations; order processing, planning,

scheduling, and accounting in manufacturing and process control; on-line and social interactions; on-line Q&A, reviews, and opinions; interactions in digital media and broadcasting; music and entertainment; on-line massive multi-player gaming etc. Therefore, a significant proportion of today's Web traffic is driven by the OLTP systems.

Figure 2.1 presents a high-level overview of an OLTP application developed in conjunction with a 3-tier MVC (Model-View-Controller) design pattern [13]. In this example, end-users' requests originate from client-side applications, such as Web browsers and desktop or mobile applications, either through HTTP (Hypertext Transfer Protocol), which is a request-reply based protocol, or REST (Representational State Transfer) [14] API (Application Programming Interface), an architectural style defining the communication approach between different computer systems. The first-tier OLTP services, which accept and handle these client requests, are stateless. Web Servers (WSs), DNS (Domain Name System) and CDN (Content Delivery Network) applications typically belong to this tier. If it is a read-only request for static content, then clients are served immediately using cached data; otherwise, update requests (i.e., insert, modify, delete) are forwarded to the second-tier services. Application Servers (ASs) handle these forwarded requests based on the application's business logics, and process the requests using in-memory data objects (if available), or fetch the required data from the underlying inner-tier services, which physically hold the data.

In Figure 2.1, an MVC (Model View Controller) pattern based OLTP implementation is considered for illustrative purposes. In an MVC application, user requests (flow **1**) (typically URLs) are mapped into *controller* actions, which then retrieve the requested data (flow **2**) using the appropriate *model* representation, and eventually render the 'view'. If in-memory representation of the requested data is not available in the object cache, then the *model* component initiates a transactional operation (flow **3**) at the inner-tier database services. Otherwise, in-memory update can take place and updated information is later pushed into the database. Note that, ASs are typically stateful and store state values (e.g., login information) as session objects into key-value stores. Inner-tier databases are partitioned and replicated based on application functionality and fault-tolerant requirements. Based on the partition placement and replica control policies, requested data tuples are retrieved (if read-only) or updated accordingly using transactional operations, and finally, the results are sent back (flow **4**) to the *model*.

The data is then encapsulated and fed (flow **5**) into the *view* to render HTML pages, and finally, sent back (flow **7**) to the user to display through *controller* action (flow **6**). Back-end tier services are later used for data warehousing, analytic, and on-line mining purposes.

Based on the high level OLTP life-cycle shown in Figure 2.1, at the very first-tier, users requests are mostly served with static contents like images, banners, and CSSs (Cascading Style Sheets). Therefore, a weak-form of converging consistency is acceptable at this level, where the end-users may even be served with *non-durable* data for an unwittingly short period of time. At the second-tier, if in-memory data object representations are already available and have not yet expired, then users are served with *semi-durable* data. The scaling application logic tier is difficult, hence developers often rely on building *soft-state* services using high scalable session and in-memory object caches in the key-value data stores [15]. However, the inner-tier OLTP databases provide strong consistency guarantee and ensure data *durability*, therefore, they are tagged as *hard-state* services. Scaling inner-tier services are also challenging, however, not impossible. Based on application requirements and functionality, an OLTP database can be partitioned appropriately to scale for serving millions of on-line clients.

2.2.1 Deployment Architecture

In recent years, with the widespread use of Cloud computing services for deploying OLTP systems, interactive and responsive Web applications need to achieve both high availability and scalability at the same time in order to serve simultaneous Web users. As shown in Figure 2.2, each tier of an OLTP application requires special attention for the required scalability, consistency, and responsiveness. A typical deployment consists of a cluster of WSs at the presentation tier, ASs at the logic tier, and Database Servers (DSs) at the persistent tier. The presentation tier needs to be highly responsive to the end-users; hence, the WSs need to be scaled-up and scaled-down on-demand as the number of requests increases or decreases.

The logic tier, on the other hand, serves *soft-state* services using session and in-memory data objects unless they have expired. Therefore, the ASs need to be accompanied by scalable key-value data stores that can support periodic growth in user

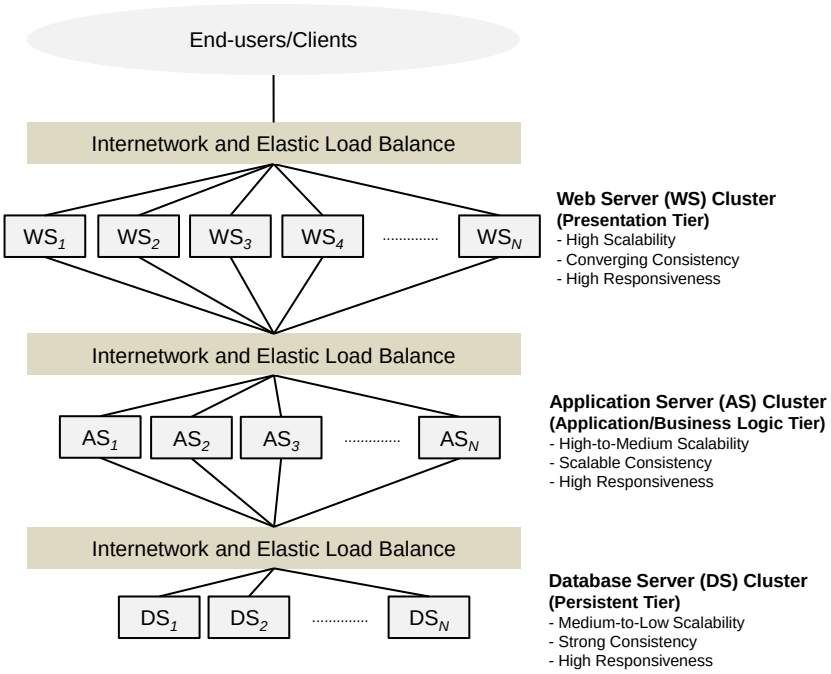


Figure 2.2: The layered architecture of an OLTP system with scalability, consistency, and responsiveness requirements

requests. Finally, the persistent tier tends to scale rather slowly compared to the former two tiers as it depends on the growth in new data arrivals. However, modern OLTP applications generate and store increasing volumes of application data and user interactions in the persistent stores to be analysed later in the analytics tier. Therefore, the underlying RDBMS services need to be highly responsive and strongly consistent at the same time, which is rather challenging to achieve in practice.

From the end-users' perspective, OLTP applications require real-time responsiveness; therefore, the underlying transaction processing systems must operate within a bounded response time. From an administrative perspective, OLTP workloads have five distinct characteristics. They are 1) short-lived transactions, 2) transactions that are highly concurrent, 3) transactions requiring very low latency from the DBMS, 4) transactions that only retrieve a small amount of data tuples, and 5) repetitive transactions, i.e., recurrence of the same transactions with different input tuples [16].

From the user interaction perspective, OLTP applications require normalised relational database schema supported by the RDBMS[1]. The deployment of such an OLTP

[1] 'OLTP system', 'OLTP database', and 'RDBMS' are used interchangeably throughout the thesis unless

database, however, depends on the following two deployment architectures:

(i) ***Shared-disk architecture*:** In an SD architecture, multiple DSs, each having private memory, share a common array of storage disks like SAN (Storage Area Network) or NAS (Network-attached Storage). Each DS, therefore, can independently serve application data requests by acting on a single collection of data. Therefore, workloads are evenly distributed within the cluster, and failover of any server is much easier to handle, as no DS has the sole responsibility for any portion of the database. Replication in an SD cluster is also easier to implement and manage. However, massive inter-server messaging overheads for locking and buffering limit the maximum number of servers an SD cluster can handle, thus limiting any high scalability guarantee. Moreover, SAN or NAS data access latency may affect end-user satisfaction and, overall, they are costly to maintain and upgrade [17]. Examples of OLTP systems following an SD architecture are Oracle RAC, Sybase ASE CE, IBM IMS and DB2 Mainframe.

(ii) ***Shared-nothing architecture*:** In an SN architecture, in contrast, each DS is equipped with one or more private storage disks, each disk having the sole ownership of the partitioned data and, thus nothing is shared with other processors. Individual Transaction Manager (TM), located in the higher-level ASs, performs data lookup through either a centralised routing table or any distributed data lookup mechanism for processing DTs. The TM collects all the required data to process a transaction locally and this process of transferring transactional data to the serving TMs is called *Data Shipping* [18]. High scalability is easier to achieve as the number of DSs in the cluster are not limited by any factor except replication. As observed in [19], a partitioned database fails to maintain consistency after replicating data over a certain factor. Examples of OLTP systems following SN architecture are Oracle 11g/12c, IBM DB2, Sybase ASE, MySQL, Microsoft SQL Server, H-Store, and VoltDB.

Figure 2.3 highlights the difference between SD and SN architectures with an example of a single database with 3 tables: 'Items', 'Customers', and 'Orders'. In an SD

otherwise indicated.

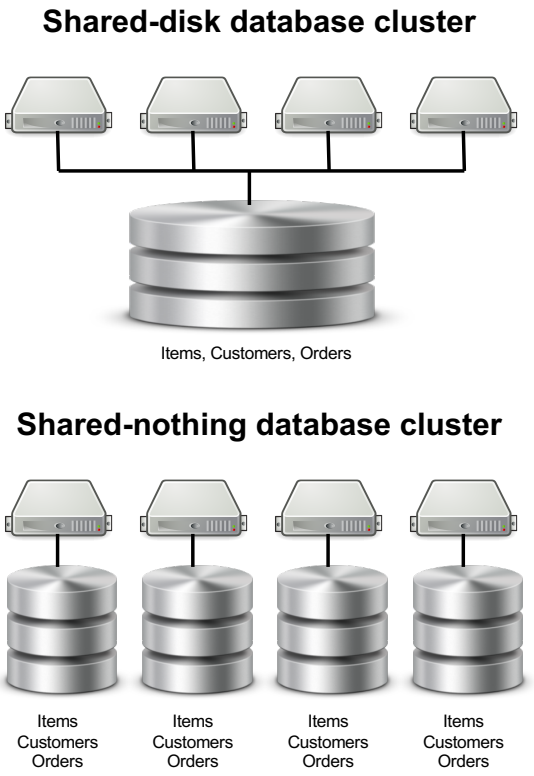


Figure 2.3: Shared-disk and shared-nothing database deployment architectures

system, the database tables are shared by multiple DSs simultaneously while for an SN system, they are partitioned within the cluster servers. Table 2.1 compares the key factors affecting the performance of SD and SN implementations of a DDBMS. In summary, despite having many challenges, SN architecture is better suited to achieve high scalability for a rapidly expanding distributed RDBMS supporting OLTP workload characteristics and requiring high concurrency control [20].

2.2.2 Transaction Processing and ACID Properties

In order to replace traditional paper-based information management and book-keeping processes, the OLTP applications need to guarantee the reliability and control over data in cases like infrastructure and network failure, eavesdropping and forging, shared access of records, and concurrent update on a single record [21]. Therefore, to manipulate the state of records, i.e., data tuples, one needs to execute *transactions*

Table 2.1: A high-level comparison of the SD and SN database deployment architectures

Challenges	**SD Architecture**	**SN Architecture**
High availability	automatic	manual failover with planned or unplanned downtime
High scalability	limited due to inter-node communications	theoretically infinite, but limited to replication factor and data shipping
Data consistency	high	Depends on data distribution transparency
Load-balance	dynamic	non-adaptable, requires repartitioning
Data shipping	none	high
Geo-distribution	difficult	easier
Bandwidth requirement	moderate	very Low
Database design	easier	difficult
Concurrency control	difficult	easier
Workload change	adaptable	requires manual intervention
Scale-up	expansive	inexpensive
Software stack	complex	easier to develop and deploy
Maintenance	minimal effort	partitioning, routing, and repartitioning

which are a collection of *read* and *write* operations on the physical and abstract application states [22]. Figure 2.4 presents a simple transaction processing architecture following XA [23] and Distributed Relational Database Architecture (DRDA) [24] standards developed by the Open Group for Distributed Transaction Processing (DTP).

A typical transaction processing system as shown in Figure 2.4 consists of three essential components: an Application Requester (AR), a Transaction Manager (TM) located at individual AS, and a Resource Manager (RM) located at individual DS. Upon receiving a data retrieval request (usually in SQL) from the application logic code, the AR component in an AS invokes its residing TM through the *TX interface* to *start* a new transaction. TM starts the transaction by locating and retrieving the required data tuples in the AS, and then sends the corresponding *read*, *write*, *update*, and *delete* com-

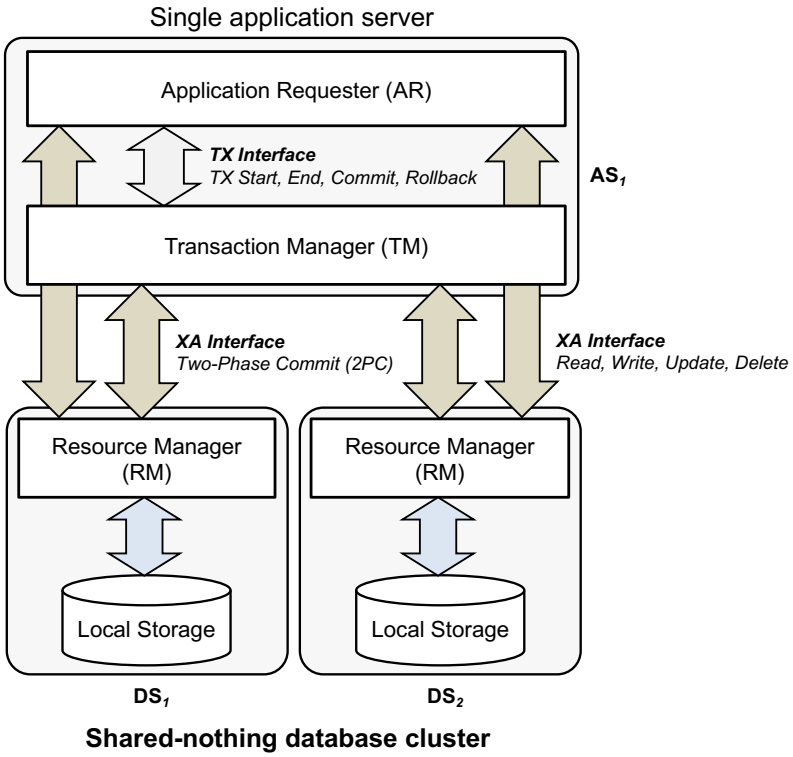


Figure 2.4: The transaction processing architecture of an OLTP system

mands to the RM using the *XA interface*. The *XA interface* between the TM and RMs also implements the Two-Phase Commit (2PC) protocol to *commit* a successfully completed transaction, or otherwise, *abort* and *rollback*. Finally, AR sends an *end* transaction command to the TM for ending the process and returns the results.

2.2.2.1 Centralised Transaction Processing

A centralised transaction processing system consists of a single DS with its local RM and local storage disks located at a single physical site. For processing transactional operations in such RDBMS, the database needs to guarantee ACID properties defined below, which are maintained by the TM:

(i) ***Atomicity*** guarantees that a transaction either executes completely and successfully commits, or otherwise aborts and does not leave any effects in the database.

(ii) ***Consistency*** guarantees that a transaction will leave the database in a consistent state before it starts and after it ends.

(iii) ***Isolation*** guarantees that to maintain the consistency guarantee, concurrent transactions will be isolated from each other.

(iv) ***Durability*** guarantees that where there are failures or network partitions, the effects of any committed transaction will not be lost.

2.2.2.2 Distributed Transaction Processing

In contrast to a standalone database system, database tables are partitioned and replicated in a distributed RDBMS with multiple DSs, each having their local RMs and storage disks. Such a system must act as a *1-Copy-Equivalence* of a non-replicated system providing ACID guarantees as below [25].

(i) ***1-Copy-Atomicity*** guarantees that a transaction is either committed or aborted at every replica where it performs the operations.

(ii) ***1-Copy-Consistency*** guarantees that consistent states are maintained across all replicas in such a way that the 'integrity constraints'—primary or foreign key constraints—are not violated.

(iii) ***1-Copy-Isolation*** or ***1-Copy-Serialisability (1SR)*** guarantees that concurrent executions of a set of transactions across the replicas are equivalent to a serial execution of that set.

(iv) ***1-Copy-Durability*** guarantees that if a replica fails and later recovers, it re-executes the locally committed transactions and also adopts the global changes committed during the downtime.

2.2.3 Distributed Transaction and Consensus Problem

A DT executes a set of transactional operations across multiple DSs and completes by a *commit* or *abort* request. The entire process is coordinated either by a single TM (as shown in Figure 2.5) or distributed TMs across multiple ARs within the AS cluster. The DT is considered committed or aborted based on the *consensus* [26] of all the participating nodes. Therefore, in a distributed RDBMS, processing DTs require running a

transaction commit protocol, i.e., a consensus protocol across the participating nodes even with the presence of a number of faulty ones.

The *consensus problem* deals with a set of nodes in a typical distributed system to agree on a specific value, decision, or an action, such that the entire system can act as a single entity. A consensus protocol must ensure three fundamental properties—agreement, validity, and termination—on the agreements or disagreements within a synchronous environment. In a DDBMS consensus, protocols are used to solve two fundamental problems: 1) distributed atomic commit, and 2) distributed concurrency control. To ensure atomic commits of DTs, two particular consensus protocols are widely used: *2PC*, and *Paxos Commit*. On the other hand, to process DTs in parallel, transactional serialisability is maintained by acquiring *distributed locks shared locks* for *read* and *exclusive locks* for *write* operations [27].

2.2.3.1 Distributed Atomic Commit

To ensure data consistency and global correctness in a distributed environment, a distributed atomic transaction needs to consider three failure modes: 1) network partition or failure, 2) failure of any site or system, 3) Byzantine failures[2] [28]. To deal with these failure scenarios, synchronous and asynchronous distributed commit protocols are proposed and used in building small-to-large scale distributed transaction processing systems. The protocols are discussed below.

(i) ***Two-Phase Commit (2PC)*:** When a DT attempts to access data from two or more replicas, *1-copy atomicity* must be guaranteed, which is implemented by the *2PC* [29] protocol. As shown in Figure 2.5(a), the 2PC protocol starts when an RM enters into the *prepared* state by sending a *Prepared* message to the TM. Upon receiving this *Prepared* message, the TM sends *Prepare* messages to all other participating RMs. When these RMs receive this *Prepare* message, they can also enter into the *prepared* state by sending a *Prepared* message reply back to the TM. While receiving *Prepared* messages from all the RMs, the TM enters into the *committed* state by sending *Commit* messages to all of the RMs.

[2] In a distributed system byzantine failure relates to faults where a faulty node or component with arbitrary behaviour prevents others from making consensus among themselves.

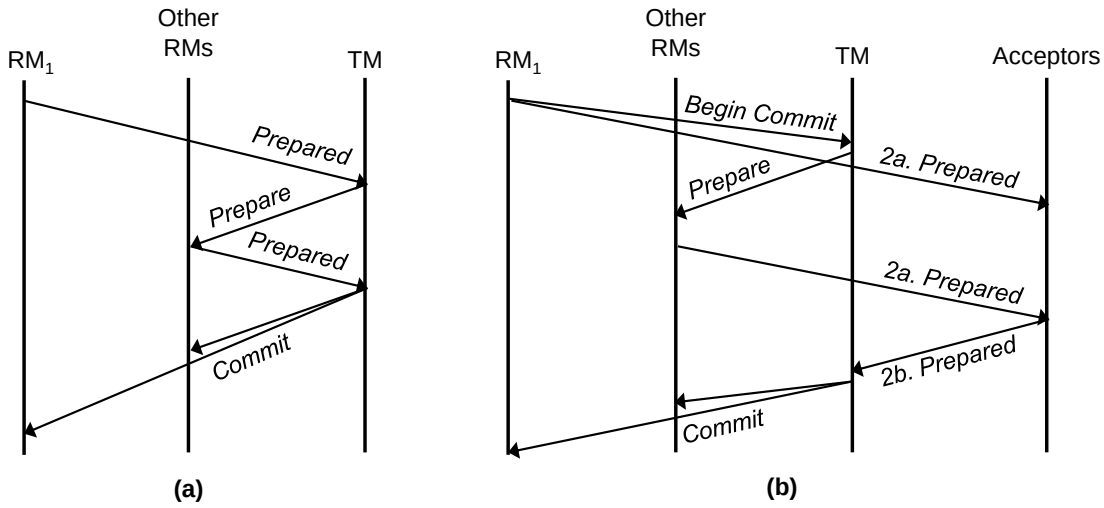


Figure 2.5: The message flow for distributed atomic commit protocols in failure-free cases: **(a)** Two-Phase Commit (2PC), and **(b)** Paxos commit

An RM can also send *Aborted* message to the TM, therefore forcing it to send *Abort* messages to the participating RMs for aborting the transactional result. During a fault-free normal case, each RM learns the *commit* or *abort* decision after four message delays and the 2PC protocol required to pass $3N-1$ messages for N number of participating RMs. Considering failure cases, if a TM does not receive a *Prepared* reply from a number of RMs, then it will *abort* the transaction. If a TM fails, the 2PC protocol *blocks* until the TM recovers. Several non-blocking versions of the 2PC protocol were also proposed (e.g., *Three-Phase Commit (3PC)* [30]) where a second TM is chosen after the first one is failed; however, none of the algorithms clearly specify what a RM will do if it receives simultaneous messages from a new TM and a recovered TM [31].

(ii) ***Paxos Commit*:** The general problem of *consensus* can be solved using the popular asynchronous Paxos [32] algorithm. This algorithm requires a collection of processes called *acceptors* to cooperate and agree on choosing a single value proposed by a preselected coordinator called *leader*. Considering a transactional system with N RMs that can tolerate up to F faulty ones, there will be $2F+1$ acceptor RMs for accepting or rejecting the vote of agreeing on the value *commit* or *abort* for a transactional execution. Therefore, the Paxos algorithm is used to

implement a fault-tolerant and non-blocking distributed transaction commit protocol [31] by letting the TM be the *leader* that proposes a *commit* or *abort* decision to be accepted by the majority of RMs. Reference [32] has proved that, despite the failure of F acceptors (i.e., RMs), $2F+1$ are required to achieve *consensus* in a distributed system.

Paxos commit runs separate instances of the Paxos protocol at each RM with the same TM to decide whether to *Prepare* or *Abort* during a transaction commit request. When an RM decides to enter into the *prepared* state, it sends a *BeginCommit* message to the TM. It then sends a *Prepare* message to all other participating RMs. If the requesting RM decides to commit, then it sends a *2a* message with the value *Prepared* and a ballot number b to all other acceptor RMs; otherwise, it sends a *2a* message containing the value *Aborted* with ballot number b. For each instance of the Paxos protocol, an acceptor RM replies with a *2b* message containing either the value *Prepared* or *Aborted* to the TM along with the corresponding ballot number b. When the TM receives at least $F+1$ reply messages from the acceptor RMs, it can then decide whether transaction needs to be committed or aborted. Finally, the TM sends a *Commit* or *Abort* message to the participating RMs. The message flows of the Paxos commit protocol are shown in Figure 2.5(b). In total $(N+1)(F+3)-2$ message passings are required to complete a Paxos commit.

Both 2PC and Paxos commit require an equal number of stable storage writes during a normal failure free case. In comparison, the TM using 2PC requires to make a *commit* or *abort* decision along with writing that decision in the stable storage. The 2PC protocol blocks for an indefinite time if the TM fails; however, the use of Paxos commit replaces the single TM in 2PC with a set of possible *leaders* and a leader can make an *abort* decision on behalf of a failed RM thereby preventing itself from blocking.

2.2.3.2 Distributed Concurrency Control

To exploit multi-processor hardware and parallel computing models, transactional executions are overlapped in an OLTP database. This requires concurrent executions

of transactions by isolating each transaction from modifying intermediate states of other uncommitted transactions. *Transaction serialisability* ensures reordering transactional operations of each transaction without changing the ordering between *read-write* and *write-write* operations on the same tuple by different transactions at the same time [33]. Simple *locking* for *read* or *write* operations at the tuple level enforces this serialisability. The *Concurrency control* [34] mechanism in a DBMS manages concurrent executions of transactions in isolation. There are two families of concurrency control protocols *Pessimistic* and *Optimistic* as discussed below:

(i) ***Pessimistic Concurrency Control*:** The *Pessimistic* approach is typically implemented using *locking*. A *shared-lock* is acquired by a transaction to get *read* access row-level locking in the database records and an *exclusive-lock* is acquired to have *write* access. If a *lock* cannot be granted by the concurrency control manager then the transaction is *blocked* in waiting state until conflicting *locks* are released. A *shared-lock* is granted if there are no other *exclusive-locks* currently hold on a record. On the other hand, an *exclusive-lock* can only be granted if there are no other *locks* currently on hold. Therefore, *read* operations are permitted to execute concurrently while *write* operations must execute serially.

Note that, *read-only* operations may also get *blocked* during a period of *exclusive-lock* holds by another transaction. Alternatively, a *write* operation may be *blocked* during a period of *shared lock* holds by another transaction. In order to ensure *strict serialisability*, all acquired *locks* are typically released only after the transaction commit or abort. The above scheme is implemented through the *2-Phase Locking (2PL)* or the *Strong Strict 2-Phase Locking (SS2PL)* protocol.

Using *2PL*, in phase-1, all required *locks* are requested and acquired step-by-step from the beginning of a transaction towards its execution. In phase-2, all *locks* are released in one step based upon a commit or abort decision. Furthermore, *deadlocks* are created as consequence of concurrent transactions *racing* to acquire *locks*. In such situations, the concurrency control manager should be able to detect such *deadlocks*.

(ii) ***Optimistic Concurrency Control*:** An optimistic approach on the other hand, allows concurrent transactions to proceed in parallel. A transaction can create

its local workspace and copy the tuples there to perform all the necessary update operations [35]. At the end of its execution, a validation phase takes place and checks whether the read-sets of the considered transaction overlap with the write-set of any transaction that has already successfully validated. If true, it has to be aborted; otherwise, it is committed successfully, by writing its changes persistently back to the database.

In case the of a DDBMS, *2PL* or *SS2PL* can also be used to guarantee *1SR*; however, there is a heavy performance penalty due to high latency in communicating with the participating nodes. To ensure *1SR*, one of the conflicting transactions has to abort in all replicas to release its locks, which then allow, the other transactions to proceed and complete their operations. Therefore, in many cases *locking* creates unwanted delays in the *blocking* state while the transactional operations could have been serialised.

Implementing distributed concurrency control through locking always creates *race conditions* locally, which may then lead to global *deadlocks*. A distributed lock manager is thus required to detect and resolve *distributed deadlocks* among the conflicting replicas using a pessimistic approach. Alternatively, to achieve global serialisation order instead of distributed locking, either a 2PC or 3PC protocol is used globally while applying 2PL or SS2PL locally. However, achieving global serialisation order is costly as it limits concurrent transaction processing and consequently reduces the overall system performance. On the other hand, an optimistic approach attempts to perform distributed or centralised conflict detection and to resolve it. However, global conflict management and serialisation are delay intensive and they are difficult to achieve in a distributed transaction processing environment.

2.3 Data Management Architectures

Two primary choices to ensure high availability and scalability for distributed OLTP databases are 1) *replication* – storing the same set of data tuples into multiple DSs, and 2) *partitioning* – splitting individual or range of data tuples into different DSs based on functional decomposition of the top-level application features or workload characteristics. Table 2.2 lists the basic strengths and weaknesses of these architectures.

Table 2.2: The strengths and weaknesses of different data distribution architectures

	Data Replication	**Data Partitioning**
Strengths	High availability	High scalability
	Workload localisation	Workload distribution
	Read scalability	*Write* scalability
Weaknesses	Data consistency	Multi-table *JOIN* operations
	Update propagation	Partition size and numbers
	Write scalability	Management complexity

2.3.1 Data Replication Architectures

Data replication is one of the most studied topics in database and distributed systems research over the last three decades. The data replication architectures typically answer three important questions: 1) 'What' is to be replicated, 2) 'How' to replicate, and 3) 'When' and 'Where' replication is done.

2.3.1.1 Replication Architectures for Distributed Computing

The existing data replication models from the distributed systems research are readily applicable in building large-scale distributed OLTP systems. In recent years, the following replication architectures have been used for building large-scale distributed data stores and reliable distributed software:

(i) *Transactional Replication*: In this architecture, a database is replicated partially or in full for distributed transaction processing. This is the most commonly practised architecture and implemented by most of the popular RDBMS providers.

(ii) *State Machine Replication*: This architecture assumes the act of replication as a distributed consensus problem and uses the Paxos algorithm to implement highly reliable large-scale distributed services [36]. For the purpose of transactional processing, deterministic finite automation is used to replicate DSs and transactional log files as *state machines*. Additionally, it provides support for distributed

synchronisation, naming services, leader election, notifications, message queue, and group membership services. It is also known as the *Database State Machine (DBSM)* approach [37]. Examples of such systems include Google's Chubby [38] lock service, Apache ZooKeeper [39], Consul [40] etc.

(iii) *Virtual Synchrony*: This architecture assumes in-memory data objects are replicated by a set of processes executing on a set of DSs in a distributed cluster. It does this by maintaining a process group where a process can join the group and can access the current state of the replicated data provided with a checkpoint. Participating processes in the group then send total order multicast[3] messages if the current state of a particular data is changed [15, 41]. Changes in the group membership are also announced using multicast messaging. Examples include Isis2 [42, 43].

As data replication is outside the scope of this thesis, only *transactional replication* architecture is discussed further in detail.

2.3.1.2 Classification Based on Replication Structure

Depending on the processing and communication overheads incurred by update propagations, data tuples are replicated in a cluster as follow.

(i) *Full Replication*: In this architecture, all tuples are replicated to all DSs in a cluster. This is typically achieved through either 1) *Master-Slave Replication*, where a master DS is responsible for processing all the incoming transactional requests and updated results are *asynchronously* copied back to the slave DS(s), or 2) *Multi-Master Replication*, where two master DSs are used to increase the transaction processing capabilities and their states are *synchronously* or *asynchronously* replicated.

(ii) *Partial Replication*: In this architecture, a subset of tuples are replicated to a subset of DSs in a cluster. There are two variants of the partial replication – *Pure*

[3] Total order multicast – if a correct process delivers message M before M', then any other correct process that delivers M' has already delivered M.

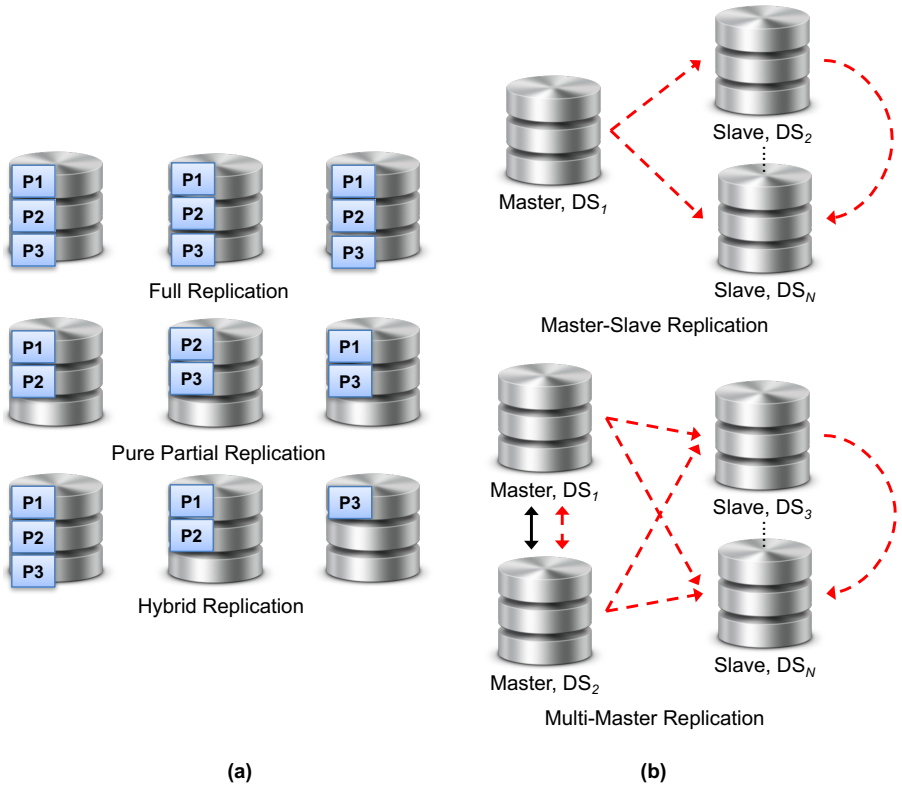


Figure 2.6: The data replication architectures – **(a)** full, pure partial, and hybrid replications, and **(b)** master-slave and multi-master replications. The black-solid and red-dashed links represent *synchronous* and *asynchronous* communications, respectively.

> *Partial Replication*, where each DS has copies of a subset of the tuples and no DS contains a full copy of the entire database, and *Hybrid Partial Replication*, where a subset of DSs contain a full copy of the entire database, while another subset of DSs act as partial replicas, containing only a fraction of the database.

Figure 2.6(a) presents the full, pure partial, and hybrid replication architectures. The master-slave asynchronous and multi-master synchronous or asynchronous replication architectures are shown in Figure 2.6(b). Note that slave DSs are commonly replicated in an asynchronous manner. Despite the cost of replication and replica management, *Full Replication* is well practised in designing a DDBMS cluster. However, with the rapid data growth in the modern OLTP environment, where relationships among the tuples are not naturally disjoint, distributed concurrency control reduces system performance and overall responsiveness [44].

On the other hand, by using *pure partial replication*, *read* operations are centralised

Table 2.3: A comparison of replication architectures based on *when* and *where* to replicate

Propagation Vs Ownership	**Eager Replication**	**Lazy Replication**	Remarks
Primary Copy	1 Transaction, 1 Owner	*N* Transactions, 1 Owner	Potential bottleneck
Update Anywhere	1 Transaction, *N* Owners	*N* Transactions, *N* Owners	Hard to achieve consistency
	Synchronous Converging consistency	**Asynchronous** Diverging consistency	

and *write* scalability can be achieved by distributing *write* operations into partial replicas. However, the benefits of partial replication come with its own shortcomings. Relational *join* operations between two database tables create DTs which limit the transaction processing capacity of OLTP systems. Furthermore, bottlenecks and hot-spots may be created in the DSs that contain the full copy of the database in *hybrid partial replication*. Lastly, workload patterns change dynamically in practice and regular schema-level changes are required to publish new application features that will involve massive inter-server physical data migrations between the DSs.

2.3.1.3 Classification Based on Update Processing

Data replication architectures are classified under two distinct aspects [44]: 1) based on the update propagation overheads – 'When' the updates propagate to remote replicas and 2) based on replica control strategies – 'Where' the *update* operations take place. Table 2.3 compares different replication architectures based on the update propagation and replica control strategies [44, 45]. The two classes of replication architectures are as follow.

(i) *Eager Replication*: In this *proactive* approach, conflicts between concurrent transactions are detected before they actually *commit*. To resolve such conflicts, updates are synchronously propagated to the remote replicas. Therefore, data consistency is preserved in spite of the cost of high communication overheads; how-

ever, this eventually increases the transaction processing latency. This architecture is also known as *active* replication.

(ii) *Lazy Replication*: It is a class of *reactive* approach that allows concurrent transactions to execute in parallel by creating local copies of the required data tuples. Each transaction makes changes in their individual local copies, therefore inconsistencies may appear between the remote replicas. Update propagations are performed asynchronously after the local transaction *commits*, and it may need to *rollback*, if conflicts cannot be resolved. This architecture is also known as *passive* replication.

Furthermore, *update* operations are processed in a replicated OLTP database either *symmetrically* or *asymmetrically*. The former requires a substantial consumption of computing and I/O resources in the remote replicas. The latter, first performs locally then only the changes are bundled together in a *write-set*, and then forwarded to the remote replicas in a single message.

2.3.2 Data Partitioning Architectures

Replicating data to a factor increases *read* scalability of a distributed OLTP database cluster. However, the ACID properties are hard to maintain once a certain replication factor is achieved even if eager replication and synchronous update processing mechanisms are used [19]. On the other hand, *write* capacities ca be still scaled through partial or pure partial replication, where only a subset of the DSs are holding a particular portion of the database. Therefore, *partitioning* (sometimes also described as *sharding*) is a commonly practised technique to split the database tables into multiple partitions or *shards* to localise *write* operations, and thus reduce the overheads of concurrent update processing.

Data partitioning can be implemented independent of replication, either by replicating the entire partitioned database at a separate geo-location (Figure 2.7(a)), or using *pure partial replication* to perform replication and partitioning in the same location, as shown in Figure 2.7(b). *Pure partial replication* has already been discussed in Section 2.3.1.2. The vertical and horizontal data partitioning techniques will be discussed here as follows:

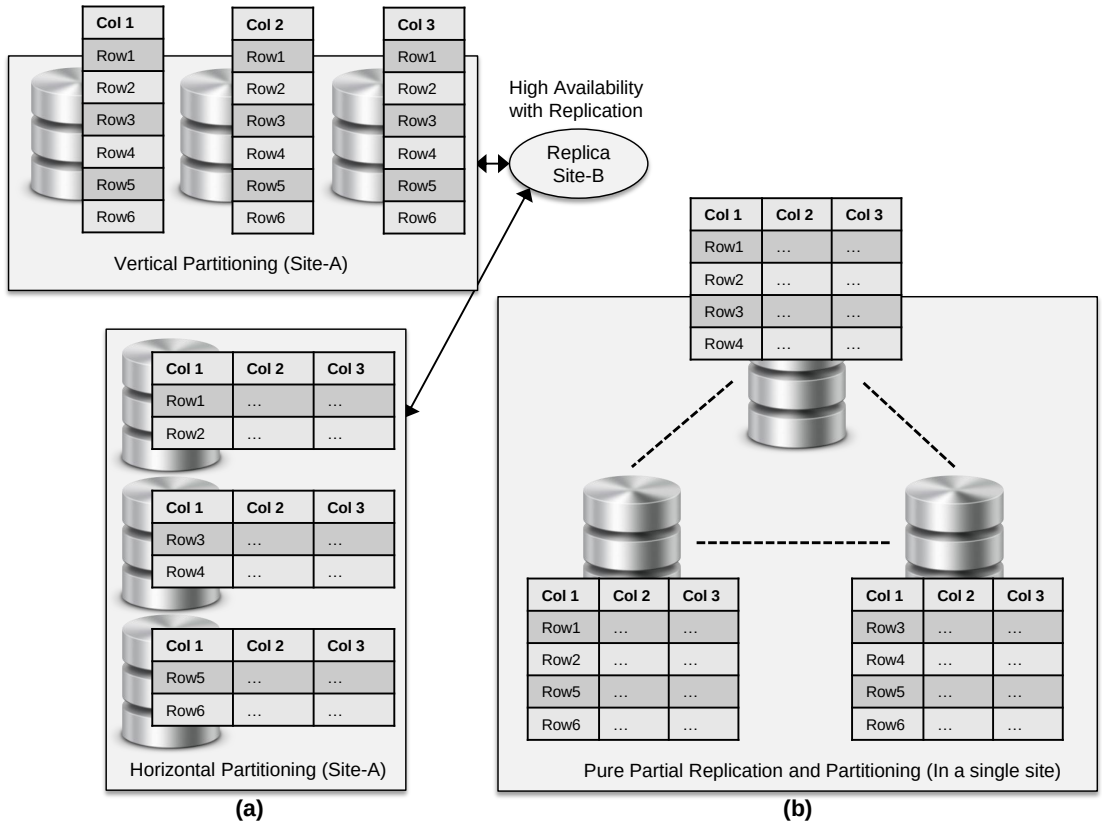


Figure 2.7: The data partitioning architectures – **(a)** vertical and horizontal partitioning, and **(b)** pure partial replication and partitioning

(i) ***Vertical Partitioning*:** As shown in Figure 2.7(a) vertical partitioning works by splitting the database table attributes (i.e., columns) and creating tables with fewer attributes. The main idea is to map different functional areas of an OLTP application into different partitions. Therefore, the achievement of scalability specifically depends on the functional aspects of a particular OLTP implementation. Hence, it is necessary to pick up the right tables and column(s) to implement the correct partition strategy. SQL JOIN operations in the RDBMS now needs to be implemented and executed in the application logic tier. Therefore, despite the simplicity of this strategy, the AS logics need a complex rewrite of *relational* operations and the scalability of a particular application-level feature is limited by the capacity of the DS hosting the partition.

(ii) ***Horizontal Partitioning*:** As shown in Figure 2.7(a) splits the data tuples (i.e.,

rows or records) across different tables based on a partition key, range, or a composition of these two. This strategy allows the database cluster to scale into any number of partitions of different sizes. The partitioning is implemented either *hash-based*, *range-based*, *directory-based*, or via a *composite* partitioning technique. Cross-table SQL JOIN operations are similarly discouraged to avoid cross-partition queries, i.e., DTs. The performance of *write* operations mostly depends on the appropriate choice of the partitioning key or range values. If partitioning is done properly, then the *partition controller* will be able to route the *write* operations in the underlying database appropriately.

2.4 Scaling Multi-Tier OLTP Applications

Recalling the overview of multi-tier OLTP application shown in Figure 2.1, each tier is responsible to perform specific functionalities and coordination between these tiers is necessary to provide the expected services to the end-users. Therefore, it is not possible to scale a single tier without providing necessary interfaces for other tiers to interact with it. As well, transactional workloads are either *compute-intensive* (require more processing capabilities and resources at the application logic tier), or *data-intensive* (require more scalability in the database tier).

In practice, regardless of the deployment and workload characteristics, data distribution logics for replication and partitioning should work in such a way that interdependencies between multiple tiers should not lead to multiple workload execution in either the database or ASs [46, 47]. Based on this analogy, there are two architectural patterns for replicating multi-tier OLTP applications as follows:

(i) ***Vertical Pattern*:** A vertical pattern pairs an AS and DS to create a unit of replication. Such a unit is then replicated vertically to increase the scalability of the entire OLTP system. The benefit of this approach is that the replication logic is transparent to both ASs and DSs, hence they can work seamlessly. However, any particular application functionality and corresponding database portion need to be partitioned appropriately across the cluster to achieve the target scalability.

The engineering costs and effort that are needed for such implementation are too high, therefore, such a pattern is rarely implemented in practice.

(ii) ***Horizontal Pattern*:** In this pattern, each tier implements the replication logic independently and requires some *replication awareness* mechanism to run between the tiers to make the necessary connections. In contrast to the previous pattern, individual tier can be scaled independently without interfering with other tiers. However, without any appropriate replication and partitioning abstractions, inter-tier interactions are not scalable without knowing how the cooperating tier is replicated or partitioned. Despite this lack of transparency, this type of pattern is common in implementing large-scale OLTP systems in practice due to their ease of deployment.

To support these two patterns, other architectural patterns are also considered such as *Replica Discovery* and *Replication Proxy*, *Session Maintenance*, *Multi-Tier Coordination*, etc. Examples of real system implementations based on these patterns are found in [47–50]. However, data distribution transparency control via multi-tier coordination is still an open research problem both in academia and industry. More recently, large OLTP service providers have developed libraries and tools to partially support such transparencies: *JetPants* [51], Gizzard [52], and Vitess [3] to name a few here.

2.5 Scaling Distributed OLTP Databases

With the increasingly widespread use of virtualisation and black-boxing at the infrastructure level services, real-time Web applications need to be designed as *highly available* and *scalable* at the same time. In recent years one of the primary techniques used to achieve these goal, is that of replacing legacy shared-nothing relational DDBMS by NoSQL distributed data stores [53]. Thus, transactional ACID properties should be sacrificed – primarily data consistency and durability for system availability, and network partition tolerance. NoSQL systems are built around the famous CAP principle [8] that was first described by Eric Brewer, and later proved as a theorem by Gilbert and Lynch [54].

According to CAP, in the face of network *Partition*, the system must choose between *Availability* and *Consistency*. This trade-off comes from the fact that, to ensure *high availability* in failure cases (i.e., crash-recovery, network partitions, Byzantine failures, etc.) the application databases need to be replicated in such a way that ensures *eventual consistency* [10]. On the other hand, data models should be simplified by eliminating table relationships (i.e., schema-free), thus there will no need to run multi-table SQL operations. Instead, application developers need to write and embed the SQL joining operations in the application logics. In short, by using legacy OLTP databases at the modern production deployment, it is hard to achieve high responsiveness without eliminating the need for a stronger form of consistency.

2.5.1 Why ACID Properties Are Hard to Scale

Based on the economy of scale, it is well known that maximising system utilisation by scaling out an interactive Web application using clusters of servers is far more cost effective than scaling up infrastructures [55]. However, deploying and managing a real-time transactional DDBMS with replication and data partitioning in a shared-nothing cluster is not trivial. With the expansion of data volume, the impacts of DTs rises to a level where managing distributed atomic commit and concurrency control using 2PC and 2PL is impractical in fulfilling Service-Level Agreements (SLAs) for responsiveness and latency.

Processing large volumes of simultaneous DTs, therefore, significantly overloads the entire transaction processing layer between the AS and DS clusters, which is eventually responsible for potential disk failures, system unavailability, and network partitions. Furthermore, with a replicated distributed OLTP DBMS, to achieve system-wide strong consistency (e.g., via synchronous update processing) requires trading-off system responsiveness and transactional throughput. Finally, in a shared-nothing system with failing components, ensuring durable transactional operations in order to meet the strong consistency guarantee is far away from reality and practice.

2.5.2 CAP Confusions

Over the past years, global trends in software development and large-scale Web data management indicates that a large group of system designers has misapplied the CAP principle to build a somewhat restrictive model of DDBMS. The narrower set of definitions presented in the proof of CAP theorem [54] seems to be one of the primary reasons. In their proof, Gilbert and Lynch considered *atomic* or *linear* consistency, which is rather difficult to achieve in a DDBMS, that is both fault and partition tolerant. Later, Brewer *et al.* [8] considers a relaxed definition of *Consistency* for the first-tier services of a typical OLTP application.

In reality, the probability of partitions in today's highly reliable data centres is rare although short-lived partitions are common in wide-area networks. So, according to the CAP theorem, DDBMSs should provide both *Availability* and *Consistency* when there are no (network) *Partitions*. Additionally, due to workload increase or sudden failure situations, the responsiveness of inner-tier database services can lag behind compared to the requirements for the first-tier and second-tier services. In such a situation, a Web scale OLTP system should be able to remain responsive and available to its clients by serving cached information until the DS cluster catches up. The acceptability of a high scalable OLTP system is to remain available and responsive to the end-users, even at the cost of tolerable inconsistency that is intentionally engineered in the application logic to hide the effects.

In his recent article [56], Eric revisited the CAP trade-offs, and described the unavoidable relationship between latency, availability and partition. He argued that a partition is just a time-bound on communication. It means that failing to achieve consistency in a time-bound, i.e., facing *P* leads to a choice between *C* and *A*. Thus, to achieve strong ACID consistency, whether or not there is a network partition, a system should both compensate responsiveness (by means of latency) and availability. On the other hand, a system can achieve rapid responsiveness and high availability within the same conditions while tolerating acceptable inconsistency.

2.5.3 BASE and Eventual Consistency

The BASE (Basically Available, Soft state, Eventually consistent) [9] acronym captures the CAP reasoning. BASE properties state that, if a system is partitioned functionally—by grouping data by functions and spreading functionality groups across multiple databases—then one can break down the sequence of transactional operations individually, and pipeline them asynchronously for update on each replicas, while responding to the end-user without waiting for their completion.

As an example, consider a 'user' table in a database which is partitioned across three different DSs by user's 'last_name' as a partition key in the following partitions A–H, I–P, and Q–Z. Now, if one DS is suddenly unavailable due to failures or partitions, then only 33.33% users will be affected, and the rest of system will still be operational. But ensuring consistency in such systems is not trivial, and not readily available as are the ACID properties in OLTP systems.

The 'E' in BASE which stands for *Eventual Consistency* [10] guarantees that, in the face of inconsistency, the underlying system should work in the background to catch up as discussed earlier. The assumption is that, for many cases, it is often hard to distinguish these inconsistent states from the end-user perspectives, which are usually bounded by different staleness criteria such as time bound, value bound, or update-based staleness. Eric Brewer had also argued against locking and actually favoured the use of cached data, but only for 'Soft' state service developments as shown in Figure 2.1.

In [57], Ken Birman has effectively shown (using the Isis2 platform [42]) that it is possible to develop scalable and consistent soft state services for the first-tier of the OLTP system, if one is ready to give up the durability guarantee. He argues that the 'C' from the CAP theorem actually relates to both 'C' and 'D' in ACID semantics. Therefore, by sacrificing durability, one can scale-out the first-tier OLTP services, while guaranteeing strong consistency and durability at the inner database tier. However, this implication of inconsistency requires a higher-level of reconfigurability, and a self-repair capability in a DDBMS, through extensive engineering efforts.

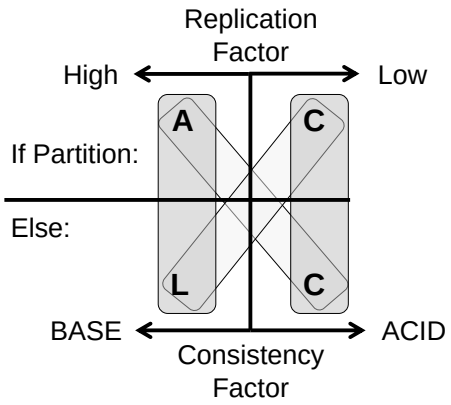


Figure 2.8: The architectural design space for large-scale distributed system development

2.5.4 Revisiting the Architectural Design Space

To overcome the confusion arising from the CAP theorem, it is, necessary to revisit the design space in the light of distributed replication and data partitioning techniques. This will also clarify the relationship between the abovementioned challenges to the ACID and BASE systems. In [58], Daniel Abadi was the first to clarify the relationship between consistency and latency for such systems. A new acronym, *PACELC*, is proposed [59], which states that – if there is a partition (P) how does a system trade-off between availability (A) and consistency (C); else (E) when the system is running as normal in the absence of partitions, how does it trade-off between latency (L) and consistency (C)?

The *PACELC* formulation is presented in Figure 2.8 under several considerations like replication factor, consistency level, system responsiveness, and partition-tolerance level. We will gradually explain them with respect to the PACELC classification for the design of a DDBMS. As explained in [60], there are four types of DDBMS one can design depending on the application-level requirements. They are described as follows:

(i) ***A-L Systems*** – always giving up consistency in favour of availability during network partition, otherwise, prefering latency during normal operating periods. Examples: Apache Cassandra [61], Amazons DynamoDB [62], and Riak KV [63] (in their default settings).

(ii) ***A-C Systems*** – provide consistent reads and writes in the typical failure-free scenarios; however, in failure cases, consistency will be sacrificed (for a limited period, until the failure recovers) in order to remain available. Examples: MongoDB [64] and CouchDB [65].

(iii) ***C-L Systems*** – provide baseline consistency (as defined by the system, e.g., timeline consistency) for latency during normal operations while, in the case of network partitions, it either prioritises consistency over availability or less responsiveness which imposes high latency. Examples: Yahoo! PNUTS [66].

(iv) ***C-C Systems*** – do not give up consistency. In the case of network partitions, these systems rather sacrifice responsiveness and latency. Examples: BigTable [6], HBase [67], H-Store [68], and VoltDB [69], etc.

Note that, completely giving up availability is not possible, as it would create a useless system. Availability actually spans over two dimensions relating to latency – 1) resilience to failures, and 2) responsiveness in both failure and failure-free cases [70]. Similarly, completely inconsistent systems are also useless, therefore the level of consistency needs to be varied from its weaker to stronger forms.

2.5.5 Distributed Data Management Aspects

Based on the abovementioned architectural design choices, four distinctive aspects are identified concerning data management in a distributed OLTP database. These fundamental aspects are necessary to understand the challenges to adopt workload-awareness in large-scale DDBMS design and development. The relevant aspects are described as follows.

(i) ***Consistency Aspect*****:** Stronger consistency models which are tightly coupled with a DBMS always ease the life-cycle of the application development. Depending on the application's requirements, giving up ACID properties in favour of BASE is also inadequate in many situations. However, stronger consistency levels are also viable to achieve by decoupling the application's logic from the underlying DBMS and implementing it along with the replica control scheme. Quorum-based

read and write query executions are one of the possible choices in this regard. Alternatively, consistency is attained in a much fine granularity [71] by ensuring entity-level or object-level consistency within a single database. Furthermore, entity-groups can be considered as a unit of consistency, and multiple groups regarded as a single unit.

A-L systems are viewed as the BASE equivalent, which tend to provide different variations of eventual consistency all the time. Similar adaption is also true while the system design choices gradually shift towards C-L systems in cases of failure. On the other hand, A-C and C-C systems by default tend to achieve stronger form of consistency either in the case of failure or not. As discussed earlier, providing ACID level consistency (i.e., serialisability) is challenging and costly to implement in a DDBMS. Therefore, providing a soft-level of consistency guarantees, such as snapshot isolation or even time-line consistency (as provided in Yahoo! PNUTS), seems to be more adaptable in such scenarios.

(ii) ***Responsiveness Aspect*:** Responsiveness is perceived by the 'delay' between initiating an application request and receiving the reply. It links two other aspects: 1) *latency* – the initial delay in starting to receive replies for a corresponding request, 2) *throughput* – the total time it takes for all the contents of a reply to be received completely. All these factors are imposed by the *Service Level Objectives (SLO)* while considering design choices.

The responsiveness of an application, AS defined by the '8 second rule' [72], outlines three response scenarios: 1) if a user action is responded to by a system within 100 ms, then this is considered to be *instantaneous*; 2) if the response is within 1 s, then the user will observe a cause-and-effect relation between performing the action and receiving the result and will, therefore, view the system as *slow*; and 3) after 8 s without a response, the user loses interest and trust in the system.

Based on this observation, A-L systems should be chosen where strict and rapid responsiveness are the requirements. Both the A-C and C-L systems are better for ensuring flexible responsiveness requirements in the face of failure and failure-free cases, respectively. C-C systems pay the costs of keeping the system up-to-

date and consistent and are therefore, less responsive when overloaded.

(iii) ***Partition-Tolerance Aspect*:** Partitions do not always arise from network or communication outages. In many cases, during busy operational hours, DDBMS are overloaded with workloads and may not be able to respond before the timeout expiration period. The possibility of *partition* is dependent highly on whether the system is deployed in a wide-area network across multiple data centres, or in a local-area network within a single data centre [73]. In choosing deployment strategies for multiple data centres distributed over WAN, one can choose between an A-L or a C-L system, due to their latency awareness during normal operationAL periods. On the other hand, A-C and C-C systems will be more appropriate to deploy within a single data centre.

(iv) ***Replication Aspect*:** The scalability of interactive OLTP systems primarily depends on how they are replicated to guarantee high *read* and *write* throughput. Increasing the number of replicas to a certain factor can scale *read* scalability up to a level after which further replication can increase the cost of update processing for *write* operations. Three types of replication strategies are typical in production deployments: 1) updates sent to all replicas at the same time (synchronous), 2) data updates sent to an agreed-upon location first (synchronous, asynchronous, or hybrid), and 3) updates sent to an arbitrary location first (synchronous or asynchronous).

Considering the above, the first option provides stronger consistency at the cost of increased latency and communication overheads, an is primarily suitable for C-C systems. The second option with synchronous update propagation also ensures consistency, but only in the case of a single data centre deployment. With asynchronous propagation, the second option provides several options for distributing *read* and *write* operations. If a primary or master node is responsible for providing *read* replies and accepting *writes*, then inconsistencies are avoided. However, it might be the source of potential bottlenecks in the case of failures. On the other hand, if *reads* are served from any node while a primary node is responsible for accepting *writes*, then *read* results can contain inconsistencies.

Both A-L and C-L systems are well suited for the abovementioned approaches un-

der the second option, as they are flexible and dynamic with latency-consistency trade-offs. Apart from preferring any node to accept *reads* and *writes*, the third option might be used as well, either in a synchronous or asynchronous fashion. While synchronous setting can increase latency, potential inconsistencies can arise using asynchronous setting. A-C and some of the C-L systems might be suit this category.

In short, interactive and real-time OLTP applications deployed over shared-nothing DDBMS aim for the following 'performance' goals: 1) *availability or uptime* – what percentage of time is the system up and properly accessible? 2) *responsiveness* – what are the measures for latency and throughput? and 3) *scalability* – as the number of users and workload quantities increase how can the target responsiveness be maintained without increasing the cost per user.

2.5.6 Workload-Awareness in Distributed Databases

A DDBMS, like MySQL Cluster [74], Google's Cloud SQL [75], Azure SQL [76], MongoDB [64], Cassandra [61], VoltDB [69], and HBase [67] that typically provide automatic replication, partitioning, load balancing, and scale-out features, only supports pre-configured partitioning rules. Under managed service deployment, these DDBMSs usually split and merge the partitions based on the number of servers (e.g., MySQL Cluster), pre-defined data volume size (e.g., in HBase), pre-defined key (e.g., MongoDB), or even based on partitioned schema (e.g., Cloud SQL). However, none of these multi-tenant managed DDBMS services are able to be adopted for dynamic workload patterns of different OLTP applications deployed by different tenants, nor can they utilise per application resource utilisation statistics in order to achieve effective load balancing. As OLTP workload characteristics are dynamic in nature, due to sudden increase in workload volumes, the occurrence of data spikes, and transient hotspots. Therefore, static data partitioning strategies failed to adopt these scenarios.

Most of the managed DDBMS services in the Cloud and standalone deployments typically use a *full* or *master-slave* or *multi-master* replication strategy, as shown in Figure 2.7. Data replication techniques have attracted much attention over the last three

decades of database and distributed systems research into ensuring fault-tolerance or high availability guarantee. In contrast, due to the recent emergence of real-time OLTP applications requiring high scalability, workload-aware dynamic data partitioning schemes are receiving much attention in recent years. Dynamic partitioning solutions are not typically available off-the-shelf, and they often require administrative interventions. These systems usually suffer from issues such as sudden workload spikes in any particular partition, hot-spotted partition or database table, partitioning storms, and load balancing problems. These are some of the potential reasons for restricted system behaviour, unresponsiveness, failures and bottlenecks. In a wide-area network setting, this leads to replication nightmare and inconsistency problems on top of added latency.

The significance of workload-aware data repartitioning is easily understood from the examples of Massively Multi-player On-line Games (MMOG) like World of Warcraft, EVE, Farmville, SimCity, and Second Life. High scalability in such environments is really challenging to be engineered, and not trivial to achieve, in contrast to other Web applications [77]. Games and virtual world users are geographically distributed and can personalise the game environments as well as make simultaneous interactions with other on-line users. Two kinds of partitioning strategies are typically observed: 1) decomposing the game or virtual world based on the application design and functionality, 2) partitioning the system based on current workload patterns [78, 79]. It is also reported that existing high scalable NoSQL data stores are incapable of handling such transactional workloads, relational schemas, and geo-distribution challenges effectively [80]. In summary, to adopt OLTP workload behaviours, it is, therefore, necessary to adopt workload-awareness in the database repartitioning logics.

In the thesis, the aim is to develop effective approaches to achieve application-level *scalability* discussed above through automating underlying database repartitioning processes. For this purpose, the related literature will be explored and a conceptual framework will be eventually developed for performing workload-aware database repartitioning. It is, therefore, necessary to develop prior backgrounds in order to understand existing repartitioning approaches properly, which will be discussed in the following section.

Table 2.4: A sample distributed OLTP database – physical and logical layouts

Servers	**Partitions**
$S_1(10)$	$P_1(5) = \{2, 4, 6, 8, 10\}$
	$P_3(5) = \{12, 14, 16, 18, 20\}$
$S_2(10)$	$P_2(5) = \{1, 3, 5, 7, 9\}$
	$P_4(5) = \{11, 13, 15, 17, 19\}$

Table 2.5: A sample transactional workload for illustrative purpose

Id	**Tuples**	**Class**
τ_1	$\{1, 4, 5, 6, 7, 8, 10\}$	DT
τ_2	$\{1, 4, 6, 9, 11\}$	DT
τ_3	$\{9, 15, 17\}$	NDT
τ_4	$\{9, 17\}$	NDT
τ_5	$\{5, 7, 18\}$	DT
τ_6	$\{15, 17\}$	NDT
τ_7	$\{2, 14, 16\}$	NDT

2.6 Workload-Aware Data Management

To understand the contributions from existing literature related to workload-aware data management in DDBMS, in this section, the basic concepts of workload network modelling, partitioning, data lookups, and general KPIs are introduced for the reader. Following this, the existing workload-aware approaches are classified and discussed accordingly. A conceptual framework is presented covering different aspects coming from the related challenges from both literature and practice. Finally, a list of existing database repartitioning schemes are compared against the proposed conceptual framework in order to reveal the shortcomings and gaps in existing literature and implementation.

2.6.1 Workload Networks

Transactional workloads contain a set of arbitrary number of DTs and NDTs, and a single transaction contains a set of data tuples. Therefore, transactional workloads can be naturally represented as a *network*[4] consisting of *nodes* or *vertices*, and their

[4] The terms *network* and *graph* are used interchangeably in the thesis. Hence, the combinations of {*node*, *link*} and {*vertex*, *edge*} are also used in the same way. In most cases, the former is used to refer the real systems, while the latter refers to their mathematical representations.

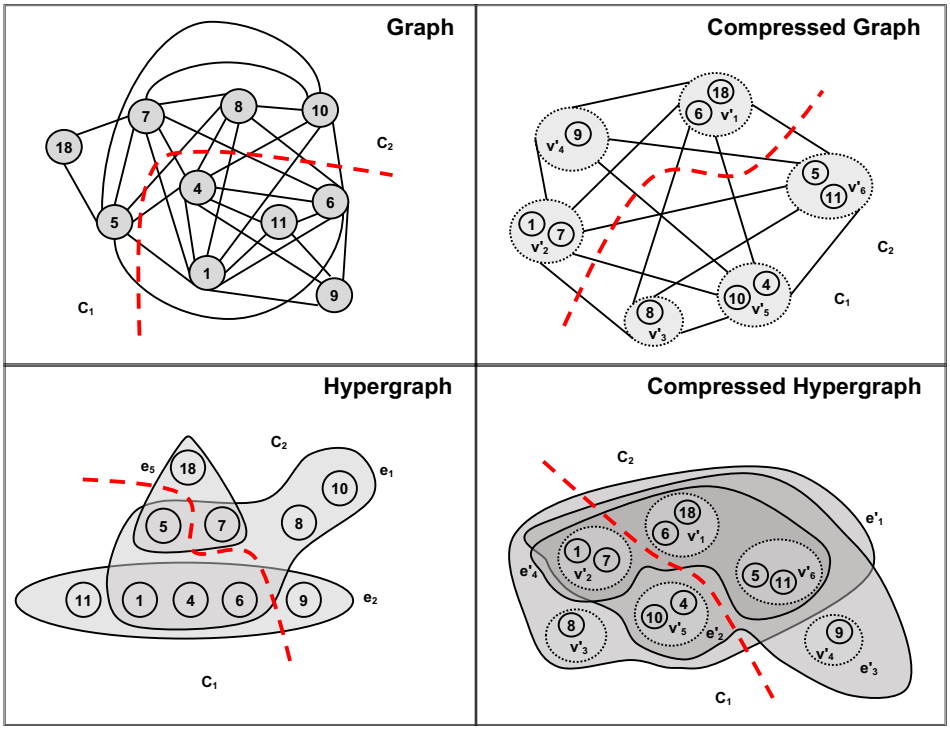


Figure 2.9: The sample transactional workload network represented as graph, hypergraph, compressed graph, and compressed hypergraph. The dashed line in *red* represents a 2-way *min-cut* of the corresponding workload network representation.

direct interactions as *links* or *edges*. Similarly, for a transaction, the set of *tuples* can be represented as a set of *vertices* connected by the *edges*, representing their interactions. For the purposes of illustration, as an example, Table 2.4 lists a sample transactional database with 20 data tuples *hash* partitioned within four logical partitions $\mathcal{P}_1, ..., \mathcal{P}_4$ residing in two physical DSs denoted as S_1 and S_2. Table 2.5, on the other hand, lists a sample transactional workload observed over time window $\mathcal{W}$ consisting of seven DTs and NDTs denoted as $\tau_1, ..., \tau_7$.

Indeed, a workload network can be represented as both graph and hypergraph, as shown in Figure 2.9, for the sample transactions from Table 2.5. In a graph representation, each *edge* links a pair of *tuples* as a pair of *vertices* for a particular transaction. If this pair of *tuples* interacts in more than one transaction, then the weight of the corresponding *edge* increases. On the other hand, in a hypergraph[5] representation, a *hyperedge* represents an individual transaction and overlays its set of *tuples* as the *ver-*

[5] A hypergraph is a generalisation of a graph that allows any of its edges i.e., *hyperedge* to connect with more than two vertices.

tices. The weight of the *hyperedge*, therefore, represents the frequency of a particular transaction in the workload. In both cases, the weight of a *vertex* represents the number of *edges* it has to other *vertices*, in other words, the number of times it has been used by a set of transactions in the given workload. This also refers to the *degree* of a *node* in the *network*. Alternatively, *vertex* weight can be denoted as the data volume size of the corresponding *tuple* with respect to a transactional database, and this latter denotation is adopted in the thesis.

Both graph and hypergraph representations can be further compressed by a factor to achieve a higher level of abstractions, thus hiding the underlying *network* level complexities. Figure 2.9 also shows the *compressed graph* and *compressed hypergraph* representations of the given sample workload. In both graph and hypergraph level compressions, multiple *vertices* are collapsed to a single *compressed vertex* by applying a mathematical operation based on the target compression level $C_l = |\mathcal{V}|/|\mathcal{V}'|$, where $\mathcal{V}$ and $\mathcal{V}'$ denote the set of all vertices in the workload and the set of target number of compressed vertices, respectively. For instance, in the example shown in Figure 2.9, $\mathcal{V}'$ is created by applying a simple modular hash function $h(k) = (k \mod C_l) + 1$ on the individual *tuple* or *vertex id* k using $C_l = 6$, which provides the *id* of the destination *compressed vertex*.

2.6.2 Clustering of Workload Networks

The existing literature, for workload-aware data management, transactional workloads are often observed over a time-sensitive observation window and collected for clustering[6] into smaller chunks containing tuples that typically appear together in the transactions. A common strategy for such an operation is to use k-way balanced *min-cut* clustering to produce k balanced clusters from an input workload network, and redistribute them within the physical DSs of a DDBMS cluster. k-way clustering follows two criteria: 1) minimum *edge-cut* guarantee, and 2) equal sized clusters, in terms of the number of vertices as well as their weights. In Figure 2.9, the *red* dashed lines represent 2-way *min-cuts* of the presented workload network representations.

[6] In the thesis, the term *clustering* refers to the partitioning of a workload network, while the term *partitioning* refers to the logical database partitioning only.

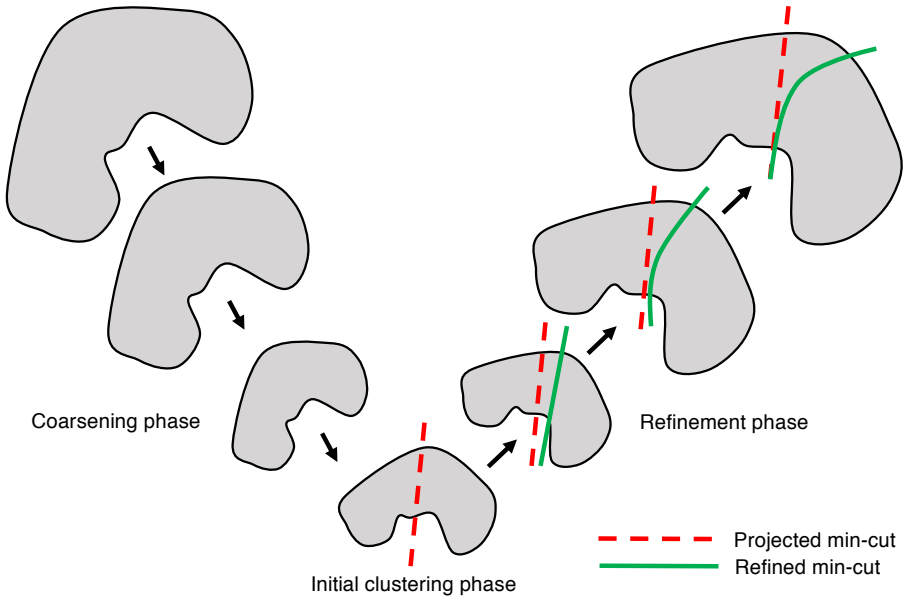


Figure 2.10: The multi-level graph clustering using the *METIS* algorithm

In the minimum k-cut or k-way *min-cut* [81, 82] problem, the task is to find k clusters by *cutting* a certain number of minimum-weight edges in a network. In other words, the goal is to find k number of cuts of minimum size in the given network. *Min-cuts* are particularly useful in divide-and-conquer type problem solving. Existing literature on workload-aware data management also utilises this well-studied approach to repartitioning an existing database based on the input workload network. Usually the order of this *min-cut* algorithm is $O(|\mathcal{V}|^2)$, and this can be further reduced to $O(|\mathcal{V}|^2 log|\mathcal{V}|)$ following Karger's randmoised *min-cut* algorithm using edge contractions [83].

In order to use such *min-cut* algorithm in real systems, a multi-level graph clustering algorithm like *METIS* [84, 85] is described in the literature (as shown in Figure 2.10). In this process, the input network is first reduced using compression (*coarsening*). The *coarsening* process is performed by a randomised maximal matching [86] algorithm [87] with the complexity of $O(|\mathcal{E}_g|)$ for an input graph containing $|\mathcal{E}_g|$ edges. Next, the smaller network is clustered, and refined back into the original network (*uncoarsening*) using the Fiduccia-Mattheyses(FM) refinement algorithm [88]. The implementation of *METIS* for both graph and hypergraph clusterings – *METIS* and *hMETIS*, respectively, is freely available in [89] and [90].

2.6.3 Data Lookup Techniques in Distributed Databases

For a DDBMS, two kinds of data lookup techniques exist – 1) centralised, using a *lookup table*, and 2) distributed. Large-scale distributed systems like HBase, MongoDB, H-Store, VoltDB, etc. use a centralised data lookup mechanism via a *lookup table* stored in a dedicated *router* node, which keeps the tuple-level location mapping of a primary key to a set of partitions. Lookup tables might also be stored as a database *index* (e.g. secondary index), and many in a existing repartitioning schemes use this idea to embed tuple locations in indices for faster data lookup. Although centralised lookup techniques are a known bottleneck to high availability and scalability for real-time OLTP application request processing, most existing database repartitioning schemes as reported in the literature as using it.

One of the oldest workload-aware frameworks, *Schism*, uses a dedicated centralised data lookup router node to handle incoming transactional requests in order to find the required tuples from the partitions. Later, Tatarowicz *et al.* [91] extended this lookup process by introducing a power router with more computational resources, and use compression techniques to optimise the record keeping for scaling up the lookup process. One of the primary challenges in using a centralised lookup process is to keep it updated following a repartitioning cycle, when a large number of data migrations occur and the lookup table needs to be updated instantly for each of the migrated tuples in order to provide consistent location information.

LuTe, another workload-aware repartitioning scheme, is proposed by [92], which primarily focuses on consistent data lookup, rather than providing inconsistent lookup results. It contains a semi-consistent lookup table which uses a temporary lookup table to keep the correct results for failed lookups, and only updates the lookup table when the data is migrated within the system's idle time. However, the above approach also lacks high scalability when thousands of lookup records need to be updated at once during repartitioning.

SWORD [93], another workload-aware incremental repartitioning approach, utilises its hypergraph compression technique, based on random hashing on tuples' primary key, as a means of handling new tuple insertion, as well as lookup existing records. The size of the lookup table for the compressed vertex-to-partition mapping, therefore

depends on the workload network compression level. Furthermore, to reduce the overheads of the lookup process, a fine-grained quorum is used to determine the required number of replicas for processing a transaction, along with a minimum set-cover algorithm to find the minimum number of partitions required to fulfil the quorum.

Distributed data lookups are primarily performed in DHT (Distributed Hash Table) [94] style data stores like Amazon DynamoDB, Riak, etc., where the initial data partitioning is created using *consistent-hashing* [2]. This is the most scalable data lookup process for large-scale data management platforms that does not have any bottlenecks. However, these data stores only support server-level entire data partition redistributions (during a scale-up, scale-down, or fault-tolerant scenario). Therefore, it is not possible to perform consistent lookups once individual data tuples are allowed to roam within the partitions during a partition-level or even transaction-level repartitioning process, as we will observe later in the thesis.

2.6.4 A Survey on the State-of-the-art Database Repartitioning Schemes

In this section, a detailed review of the existing DDBMS repartitioning schemes, related to the objectives in the thesis, will be presented. We derived two classes of database repartitioning strategies from the existing literature – 1) *static* schemes, which primarily pay attention to automate database design, increase I/O parallelisms, and scale transactional processing with the transient change in workload; and 2) *dynamic* schemes, which primarily focus on adopting workload dynamics, automatic database administration, and incremental repartitioning. A number of existing schemes will be discussed below, based on this abovementioned high-level classification, in order to identify their contributions and specific shortcomings.

2.6.4.1 Static Repartitioning Schemes

In this section several state-of-the-art static repartitioning schemes are discussed under specific categories.

2.6.4.1.1 Database Schema-aware Static Repartitioning

One of the oldest techniques for intelligent data placement in a shared-nothing OLTP database is proposed in [95]. Several algorithms are presented to utilise the *degree of declustering* in a transactional workload while deciding on the placement of partitioned data in a cluster. Based on the database schema design, transactional tuples are either fully or partially put together in a single or multiple DSs to increase inter-server transaction processing parallelisms. To handle small tuple-level relations that can not be equally partitioned, three specific strategies are evaluated – 1) *random*, 2) *round-robin*, and 3) *Bubba* [95] – where tuples in a relationship are placed in decreasing order of their access frequency. For a non-partitioned workload, two strategies are proposed – 1) full declustered relationships are partitioned across all the DSs, and 2) partially declustered relationships are partitioned over a subset of DSs. Finally, for transactions that can be partitioned, the full declustering option is chosen to increase transactional parallelism. However, the effects of fully or partially converting the DTs into NDTs based only on the database schema relation, during the repartitioning cycle, are not investigated. Therefore, it is not possible to determine any adverse effects on server-level data distributions and overall database performance from the off-line data migrations.

2.6.4.1.2 Database Design-aware Static Repartitioning

To automate a physical database design for a single node Microsoft SQL Server, Agrawal *et al.* [96] have proposed including workload characteristics and partitioning schemes during the design phase. A similar approach is taken to include database partitioning design advisor for IBM DB2 OLTP databases [97] which, for a given workload, advises physical database design attributes like indexing, views, table partitioning, and clustering of multiple dimensions. A similar approach is also presented by Nehme *et al.* [98], where a partitioning advisor is proposed that advises table-level replication and partitioning decisions based on workload patterns in order to optimise future queries. The authors have proposed the Memo-Based Search Algorithm (MESA), a partitioning advisor algorithm, which is deeply integrated with Microsoft SQL server's distributed query optimiser for constant workload evaluation. However, all of these proposed schemes involve searching through a large number of physical design choices, and providing

efficient static partitioning decisions for the initial database design. Therefore, the proposed schemes are unable to adopt with dynamic workload changes, as it requires altering the database design over and over again, which is not realistic in practice.

2.6.4.1.3 Graph Theoretic Static Repartitioning

Kallman *et al.* [99] design and implement an in-memory shared-nothing OLTP DDBMS called *H-Store*. One of the primary assumptions from their analysis of popular OLTP workloads reveals that most of the databases have a *tree* shaped schema, with a *root* table on the top with $n-1$ relationships with the other tables. Therefore, such a *root* table can be easily range- or hash-partitioned on its primary key. This will distribute the workloads evenly among the DSs in the cluster to reduce the number of DTs. However, as shown in [100], the modern OLTP database can have all shapes of schemas such as tree, star, or snowflake. Therefore, static data partitioning strategies no longer stay valid for extended operational periods, as workload patterns change.

Schism [101] is the first of its class of workload-aware static repartitioning schemes that uses graph-cut techniques to reduce the proportions of DTs in the current workload. The main idea of Schism is to represent a given workload sample as a graph, where vertices represent the tuples and edges represent the transactions. Later, the external graph *min-cut* library, *metis* [89], is used to create balanced clusters of tuples, while minimising edge-cuts. Replication is also applied by exploding each vertex to a star shaped configuration containing $n+1$ vertices, where n is the number of transactions accessed by a tuple represented by that vertex. The use of replication simplifies the overall network dynamics, as a DT can be served as a NDT with its replicas residing in the same DS. Clusters are then placed into separate DSs, and a predicate-based classification tree is formed, which then direct future transactional requests to the appropriate DS. Since then, *Schism* has been integrated into *Relational Cloud* [102], which is a prototype implementation of a managed DDBMS service in the Cloud.

The primary shortcoming of Schism is that graph workload networks are unable to fully represent transactional relationships, since they present a fine-grained representation of the OLTP workload. Furthermore, by using replication and partitioning together, the network complexities are somehow abstracted away from the external

graph-cut library, which needs to create proper clusters to reduce the impacts of DTs in the system effectively. Finally, this static approach neither adopts well to workload changes, nor can it scale with the increased size of the input graph or properly handle replication related data inconsistencies.

2.6.4.1.4 Skew-aware Static Repartitioning

A skew-aware automatic database partitioning approach, along with an automatic database design tool called *Horticulture*, integrated with *H-Store* [68], is proposed in [16]. The proposed work utilises *Schism*, and aims to mitigate the effects of a temporally skewed workload by redistributing the data and workload distributions, which eventually minimises the percentage of DTs in the system. *Horticulture* adopts an automatic database partitioning algorithm based on neighbourhood search, on a large solution space of possible database designs considering four possible options – either horizontally partition a table, replicate, create secondary indexes, or create routing for the execution of stored procedures. The authors also provide an analytical cost model for estimating the coordination cost C_c of processing a DT and skewness for a given workload sample as below:

$$C_c(\tau_{d_i}) = \left(\frac{|S_{\tau_{d_i}}|}{|T||S|}\right)\left(1+\frac{|T_d|}{|T|}\right) \tag{2.1}$$

where $|T|$ and $|T_d|$ are the number of transactions and DTs, respectively, and $|S_{\tau_{d_i}}|$ is the maximum number of DSs a DT is spanned across. The skew factor $\mathcal{S}_k$ is calculated as follows:

$$\mathcal{S}_k = \frac{\sum_{i=0}^{n} \mathcal{S}_{k_i}|T_i|}{\sum_{i=0}^{n}|T_i|} \tag{2.2}$$

where $\mathcal{S}_{k_i}$ is calculated using the algorithm given in [16], and n is the number of observation intervals. Finally, given a database design D_d and a sample workload $|\mathcal{W}|$, the cost of choosing $\mathcal{D}$ is defined as follows:

$$c(D_d, |\mathcal{W}|) = \theta C_c + (1-\theta)\mathcal{S}_k \tag{2.3}$$

where $0 \leq \theta \leq 1$ is the exponential averaging coefficient to prioritise either the coordination cost of a given DT or the workload skew.

The major drawback in this approach is that of using a specific snapshot of the sample workload in order to calculate the coordination costs of the DTs and skew factors, in order to choosing a particular database design to implement, as a temporary solution for a period of time. Therefore, this approach is neither adaptable to workload changes, nor is it scalable to implement a desire solution on-the-fly without performing off-line data migrations.

Based on the research of [16, 101, 102], Jones *et al.* [103] developed *Dtxn* – a fault-tolerant distributed system design tool, which is first in its class for designing OLTP applications with a shared-nothing DDBMS, supporting live data migration for workload-aware static data repartitioning. Turcu *et al.* [104] also utilise *Schism* and *Granola* [105] frameworks to combine the benefits of both approaches. To increase parallel transactional executions, *Granola* utilises timestamped coordination to serialise NDTs without locking, on the assumption that the DBMS is well partitioned beforehand. Furthermore, it presents an automatic framework that uses the partitioning layout produced by *Schism*, and then employs a machine learning based routing technique to support parallel executions of the NDTs. However, this strict assumption is rarely observed in practice. Therefore, none of the abovementioned static schemes can perform effective repartitioning without taking the database off-line. Moreover, without incrementally running the repartitioning process, it is not possible to dynamically adopt to the transient changes in OLTP workload.

2.6.4.2 Dynamic Repartitioning Schemes

In this section several state-of-the-art dynamic repartitioning schemes are discussed under specific categories.

2.6.4.2.1 Dynamic Repartitioning for Elastic Transaction Processing

The most important characteristic of a dynamic workload-aware repartitioning scheme is to ensure elastic transaction processing by continuously redistributing on-line data tuples time to time without impacting the ongoing database operations. Workload-aware data partitioning and placement, elastic transaction processing, and live data migration techniques are extensively studied in [100]. Four primary design principles

are mentioned in [100] for dynamic repartitioning scheme design – 1) a separate system state from the application state, 2) data storage decoupled from ownership, 3) limit common operations to a single site, 4) limit distributed synchronisation.

The authors have proposed partitioning the database based on transactional access patterns, where a key grouping protocol, *ElasTraS* [106, 107], is implemented, which dynamically co-locates data ownership within a partition that eventually reduces the number of DTs in the workload. *ElasTraS* uses a dynamic partitioning protocol which automatically splits and merges data partition, as well as replicating global tables. The main idea is to use a *key* group abstraction model which allows an application to specify which the data tuples on-the-fly that it wants for transactional access [106].

A live data migration technique, *Albatross* [108], is also proposed, which identifies four primary associated costs – 1) service unavailability, 2) the number of failed requests, 3) the impact on response time, and 4) data transfer overheads. At first, the live migration procedure involves migrating the cache data for running transactions, which reduces the impacts of transactional latency. Next, the states of migrating transactions are copied from the source to the destination partition, where the associated data items have been already migrated.

To migrate the entire logical partition between the DSs, a partition migration technique, *Zypher* [109], is proposed, which performs off-line partition migration by stopping all transactional executions and freezing any structural changes in the schema. Detailed evaluations of both synchronous and asynchronous data migrations are performed by the authors in [110]. Elmore *et al.* have proposed *Squal* [110, 111], which similarly provides efficient and fine-grained live reconfiguration of partitioned OLTP DDBMS. In *Squal*, a distributed reconfiguration process starts by any designated DS in the cluster which has been affected by load imbalance. A global lock is acquired among the DSs participating the reconfiguration process, followed by either a reactive or asynchronous migration.

Chen *et al.* [112, 113] have proposed ***S**cheduling **O**nline D**a**tabase Re**p**artitioning (SOAP)*, which primarily aims to reduce the repartitioning time and cause fewer impacts to the ongoing transactions. *SOAP* mixes repartitioning operations with normal transactional executions with two distinct approaches – 1) by prioritising repartitioning

transactions in the processing queue, and 2) by scheduling repartitioning transactions at the system idle time. A feedback control loop is combined with the proposed framework to determine the orders and frequencies of the repartitioning transactions for optimising the scheduling decisions. The repartitioning decisions can come from the automatic database design tool or directly from the database administrator.

Although the abovementioned dynamic repartitioning schemes can support elastic transaction processing in a graceful manner, due to the complexities involved in identifying the ownership of data tuples from individual DTs, they are not practical to use for OLTP applications in real-time. OLTP workloads often need simultaneous repartitioning of the underlying database to reduce the adverse impacts of DTs within the system. In most cases, it is, therefore, necessary to make proactive data placement decisions, in order to avoid potential bottlenecks in scalable transaction processing.

2.6.4.2.2 Dynamic Repartitioning for Social Network Workloads

Pujol *et al.* [114, 115] utilise graph *min-cut* based clustering for partitioning On-line Social Network (OSN) databases by combining both replication and partitioning mechanisms in a middleware called *SPAR*. Social network relationships between the end-users are represented by graph, then graph *min-cut* clustering is used to create clusters containing users who are neighbours of one other. Finally, the clusters are placed into separate DSs to reduce the span of frequently occurred queries. However, dynamic user level interactions in a social network are not considered in *SPAR*; therefore, it is unable to reduce the impacts of DTs, deriving from end-users' interactions in an OSN.

This limitation is captured in [116] by including both the OSN structure and user interactions when constructing hypergraph networks for graph *min-cut* clustering. It includes multi-way social interactions between the users, and creates a temporal activity hypergraph, where a hyperedge represents an activity transaction and the vertices represent the tuples accessed by it. The proposed framework is evaluated under a *Twitter* workload in a Cassandra [61] cluster, which shows good adaptability of the graph *min-cut* based workload-aware data repartitioning for a social network workload.

Database repartitioning strategies for OSN workloads are also studied in [117], where two primary aspects of workload-aware repartitioning are addressed – 1) how

to reduce the latency of DTs for following a Zipfian [118] distribution in the workload, and 2) high temporal locality of DTs. To address the first challenge, a *Two-Tier Hashing* technique is used to identify the vertices with large degree distributions, and distribute them within the DSs, in such a way that increases the parallelism for large fan-out transactions, while preserving the vertices with smaller degrees locally. For addressing the second challenge, a workload-aware repartitioning scheme is proposed which attempts to keep the vertices that often interact with each other, in the same physical DS, thus reducing the number of DTs.

The proposed partitioner in [117] distributes transactional edges connected with a popular vertex within a fixed number of DSs to control the transactional fan-out. Furthermore, the partitioner derives a special sub-graph from the workload network that is most active, and only uses it for clustering purposes using *METIS* [89]. The primary shortcoming in this scheme is that the fan-out calculation process does not consider the server-level data distributions, nor does it control the amount of data to be migrated over the network during each repartitioning cycle. Therefore, with the high volume of OSN workload, it will be challenging to process the transactions concurrently while, at the same time, keeping the frequently occurring tuplesets together when the DSs become overloaded and are busy performing large volumes of data migrations.

2.6.4.2.3 Graph Theoretic Dynamic Repartitioning

***H**yper-graph based **O**LTP database **P**artitioning **E**ngine (HOPE)*, which is a hypergraph based dynamic repartitioning scheme is presented in [119–121]. For a given workload, it constructs a weighted hypergraph, and then iteratively performs network clustering in order to produce feasible and near-optimal clustering decisions. Following each iteration, *HOPE* evaluates the clustering feasibility, takes administrative feedbacks, and then either performs data repartitioning or a hypergraph refinement. *HOPE*'s unique workload analysis involves constructing a *minterm predicate*[7] predicate list with corresponding transactional access counts, which are then used to construct a weighted workload network. Finally, *hMETIS* [90] graph *min-cut* library is used to cluster the hypergraph and any other refinements incrementally. For feasibility analysis of the

[7] A minterm predicate consists of the set of all predicates with conjunctions and negations that are used to describe horizontal fragments of a relation in a DDBMS.

output decisions, a combined value of data and workload skews, resulting for each of the given clustering outcome, is calculated as follows:

$$\mathcal{S}_k = \frac{1}{n}\sum_{i=1}^{n}\left(\theta\left(|\mathcal{D}_{S_i}| - \frac{1}{n}\sum_{i=1}^{n}|\mathcal{D}_{S_i}|\right)^2 + (1-\theta)\left(|T_{S_i}| - \frac{1}{n}\sum_{i=1}^{n}|T_{S_i}|\right)^2\right) \tag{2.4}$$

where $|\mathcal{D}_{S_i}|$ and $|T_{S_i}|$ are the size of the ith cluster (i.e., data volume size of the ith destination server) and the number of transactions accessing it, respectively. $0 \leq \theta \leq 1$ is the exponential averaging coefficient to prioritise between data and workload skews.

Shang *et al.* [122] investigate workload network behaviours in graph databases by exploring the change in observation window sizes over time to find a better graph-cut based dynamic repartitioning scheme. As graph algorithms do not typically access all the vertices in a given workload network, the static graph repartitioning scheme is unable to achieve a good workload balance within the DDBMS cluster. Nine popular graph algorithms are categorised under three groups to examine using the proposed framework. The primary observation from their study is that, by utilising the frequently accessed tuplesets, it is possible to use the previous observation window as the basis for creating a graph superset in the current window. The experimental results show that mining the most frequently accessed tuplesets from the consecutive observation windows is sufficient to build a high level representative graph network which is just sufficient to produce high-quality data repartitioning decisions.

2.6.4.2.4 Incremental Dynamic Repartitioning

Elasca [123], yet another dynamic repartitioning scheme, utilises a workload-aware optimiser that predicts efficient partition placement and migration plans. The optimiser minimises the computing resources required for elastically scaling-out the cluster, and maintains server-level load-balance. Given the current system statistics, the optimiser finds an appropriate assignment $\tilde{A}$ for partitions to redistribute them among the DDBMS cluster after a new DS is added or an existing DS is removed as below:

$$\min_{\tilde{A}_{ij}} \quad \theta\sum_{\forall j}\sum_{\forall i \forall t}\left(|\tilde{A}_{ij}^{t+1} - \tilde{A}_{ij}^{t}||\mathcal{P}|_i^t\right) + (1-\theta)\sum_{\forall j}\left|\left(\sum_{\forall i \forall t}\tilde{A}_{ij}^{t+1} l_{\mathcal{P}_i}^t\right) + l_{S_i}^{t+1} - \tilde{l}_{S_i}^{t+1}\right| \tag{2.5}$$

where $|\mathcal{P}|_i^t$ and $l_{\mathcal{P}_i}^t$ are the size and the load generated by partition i and at time t respectively, and $l_{S_i}^{t+1}$ and $\hat{l}_{S_i}^{t+1}$ represent the projected average overhead and CPU load on a single DS, respectively, at time $t+1$. The first part of the above equation minimises data migrations whereas the second part represents the load per DS. $0 \leq \theta \leq$ controls the balance in optimising between data migrations and server-level load-balance. *Elasca* is evaluated by implementing as a component within *VoltDB* [69], where it shows effectiveness in a parallel OLTP execution environment. However, *Elasca* treated the workload as a *black-box* and did not dive deeper to exploit transaction-level optimisations while migrating partitions within the cluster. Furthermore, it did not consider minimising the number or impacts of DTs.

Kumar *et al.* [93, 124, 125] have proposed *SWORD* – a workload-aware incremental data repartitioning framework. To the best of our knowledge, it is the first scheme which performs exhaustive incremental data migrations, as the workload changes, in order to reduce the number of DTs under a target threshold. The proposed algorithm represents the transactional workload as a compressed hypergraph network, where the vertex ids are hashed by their primary key to collapse as a compressed vertex, and hyperedges linking those compressed vertices form compressed hyperedges. By abstracting the workload network at a coarser level, and then utilising graph min-cut to create balanced clusters, thereby further minimising edge cuts, this in turn reduces the percentage of DTs in the system. After this one-time graph-cut based repartitioning, selective compressed vertex pairs are swapped between the DSs to maintain the initial gains achieved by the *min-cut* as well as server-level load-balance. At every iteration, when the percentage of DT reaches a triggering threshold, a priority queue is formed with the compressed hyperedges ordered by their maximum contributions in the cut, which is defined as C_e. Pairs of candidate compressed vertices are preselected for swapping between a pair of DSs based on the swapping gain. Let $\mathcal{S}_g$ be the swapping gain within a DS pair (X,Y) defined as follows:

$$C_e = \frac{w(E'_h)}{\sum_i w(e'_{h_i})} \tag{2.6}$$

where $w(E'_h)$ is the weight of ith compressed hyperedge in the cut, and the weight value

is represented by the sum of frequencies of the hyperedges reside in i.

$$\begin{aligned} \mathcal{S}_{g_{X\rightarrow Y}} &= w(E'_h) - \left(\sum_{i \in Y} w(e'^{\bowtie}_{h_{i_Y}}) - w(E'_h) \right) \\ &= 2w(E'_h) - \sum_{i \in Y} w(e'^{\bowtie}_{h_{i_Y}}) \end{aligned} \tag{2.7}$$

where $w(e'^{\bowtie}_{h_{i_Y}})$ is the sum of weights of compressed hyperedges, incident on compressed vertex e'_{h_i}, in server Y. Swapping gains are calculated for each compressed hyperedge from the queue, and exhaustive checks made as to whether swapping a particular pair can achieve any gains in terms of reducing the percentage of DTs in the overall system. Although the exhaustive incremental repartitioning algorithm in *SWORD* aims to incrementally decrease the percentage of DTs in a system, it does not guarantee progress in cases where no suitable pairs are found to achieve a swapping gain. Furthermore, the solution only relies on the initial *min-cut* result, which changes over time as the workload changes, and at each iteration the algorithm has to exhaustively search through a large solution space, which makes it difficult to apply in practice.

2.6.4.2.5 Dynamic Repartitioning for In-memory OLTP Databases

More recently, Taft *et al.* have developd *E-Store* [126], which provides a two-tier elastic data partitioning support for *cold* and *hot* data tuples within an in-memory OLTP DDBMS. *E-Store* uses a two-phase monitoring technique where, in phase-1, system level metrics are monitored until they have reached a certain threshold, then phase-2 is initiated. In phase-2, tuple-level monitoring is started which reveals the top k frequently accessed tuples as the *hot* data objects. Next, a two-tier *bin packing* [127] algorithm is used to generate the optimal placement of tuples to the partitions. It calculates partition-level loads, in terms of the access counts for *hot* and *cold* data, followed by finding optimal placements for *cold* tuples in blocks to partitions, and small clusters of *hot* tuples to targeted partitions. The assignment of tuples to partition is performed via one of three distinct ways – 1) a *greedy* approach assigns *hot* tuples to partitions incrementally which are less loaded; 2) a *greedy extended* approach first applies the *greedy* scheme, and if the cluster is still imbalance, then *cold* blocks from the imbalanced partitions are redistributed to the partitions with lower loads; and 3)

a *first fit* approach mixes up *hot* and *cold* tuples together in a single partition which, in some situations, leads to better resource utilisation. *E-Store* utilises *Squal* [110] for live data migration purposes. Experimental results show that *E-Store* is capable of handling transient hotspots, time varying skewness, sudden workload spikes, and hockey stick effect [128], as well as perform database reconfiguration faster then other NewSQL [129] systems. Serafini *et al.* have developed *Accordion* [130], a dynamic data placement scheme for in-memory OLTP DDBMS, by considering the association between partitions which are transactionally accessed together more frequently. Based on the estimation of server-level load statistics and associations on the maximum throughput, *Accordion* finds partition placements which maintain server-level load-balance, minimise inter-server data migration, and minimise the number of servers involved. The partition placement problem is formulated as Mixed Integer Linear Programming (MILP) problem and always find a solution if the problem size is feasible. However, none of these abovementioned schemes make decisions based on actual repartitioning KPIs that have proper physical interpretations, and directly relate to the key challenges in reducing the impacts of DTs with minimum skew in data distributions.

2.6.4.2.6 Dynamic Repartitioning for Automatic Database Management

With the gaining popularity of shared-nothing OLTP DDBMS, large-scale OLTP service providers have developed their own implementations of dynamic repartitioning libraries for automatic database management. Most of these toolkits use shared-nothing MySQL DSs and are not workload-aware (i.e., not continuously analysing workload dynamics for repartitioning decisions). *JetPants* [51], which was originally developed by *Tumblr* – a social networking platform, is an automatic tool to manage high availability, partitioning management, load balancing, and repartitioning, and lookup. It primarily supports *range-partitioning* for standalone MySQL system. *Vitess* [3, 131] provides support for automatic partitioning of shared-nothing MySQL DDBMS, and is currently used to handle *YouTube*'s video streaming workload. Apart from providing features like high scalability, automatic database connection pooling, automatic failovers and backups, it also assists in on-line repartitioning and data migrations with minimum read-only downtimes. *Gizzard* [52] is a similar middleware, originally developed by *Twitter* to manage automatic data partitioning, elastic migrations, and partition-level replication

through a declarative replication tree across a range of shared-nothing RDBMSs. It uses a forwarding table to store key-partition mappings for data lookup, and provides an *eventual consistency* guarantee for transactional operations. Finally, *MySQL Fabric* [132] is a recent development by *MySQL* to support high availability and scalability requirements for real-time OLTP applications in the Web. Apart from providing automatic failure detection and transparent failover, it supports administratively defined repartitioning and load balancing across multiple MySQL server groups for *range-* and *hash-based* initial data partitioning.

2.7 A Workload-aware Incremental Repartitioning Framework

Based on the discussion above, a conceptual framework for workload-aware incremental repartitioning is constructed which should cover the important aspects, described below, drawing directly on the existing literature. Based on these aspects, the repartitioning scheme should be developed and configured in such a way that it satisfies the actual purpose of the particular transaction processing system. Later, we will show to what extent the repartitioning approaches, discussed above, can fulfil these aspects, and this will help us to understand the gaps in existing research as well as a way forward to address them. The different aspects of the conceptual framework are presented as follows:

(i) ***Design Aspects****:*

- *Transparent (TRS)*: The repartitioning process is entirely independent from the underlying DDBMS and operates without interfering in the database operations, therefore entirely abstracting the repartitioning complexities.
- *Shallowly-integrated (SIN)*: The repartitioning process can leverage the underlying DDBMS architecture for effective decision making by shallowly integrating the repartitioning framework to the database cluster.
- *Deeply-integrated (DIN)*: The repartitioning framework is implemented as an integral component of the DDBMS where transactional execution directly relies on its operational capability.

(ii) ***Granularity Aspects**:*

- *Level of abstractions in repartitioning approaches (Implicit (IMP)/Explicit (EXP))*: The repartitioning framework can follow either an implicit workload clustering based approach using graph *min-cut* techniques, or an explicit heuristic based approach for finding approximate or near-optimal outcomes.
- *Level of abstractions in workload representations*: For graph *min-cut* based repartitioning, the workload networks can be constructed based on the level of abstractions required to understand the tuple-level transactional interactions. Networks can be represented as Graph (GR), Hypergraph (HR), Compressed Graph (CGR), and Compressed Hypergraph (CHR). Compressed to simplified abstractions provide coarse-to-fine–grain workload granularity.
- *Level of abstractions in data migration approaches (Partition-level (PDM)/Tuple-level (TDM)*: Depending on the construction and level of abstractions of the repartitioning framework, the data migration process can take place at different levels, where higher-level (cluster-to-server) abstractions can minimise the physical data migration costs, and lower-level (partition-to-partition or cluster-to-partition) abstractions achieve better data or workload distribution balance.

(iii) ***Performance and Optimisation Aspects**:*

- *Impacts of distributed transactions (DT)*: The primary repartitioning objective for an OLTP system is to reduce the number of DTs in the workload; however, without physically interpreting how DTs are impacting the system in a rational way, it is difficult to built a novel repartitioning framework which can incrementally reduce the overall impacts of DTs over time.
- *Load balance (Server-level (LBS)/Partition-level (LBP))*: The framework should be able to balance data tuple distribution, as well as workloads, access at both server- and partition-level. Without balancing partition-level distribution, data and partition migration techniques will not be able to play effective roles in practical deployments.
- *Data migrations (DM-ON/DM-OFF)*: A repartitioning framework must provide effective means for supporting both on-line and off-line data migrations based

on the actual requirements, while minimising the cost of transaction throughput, latency, and network resource consumption.

(iv) ***Management Aspects**:*

- *Data lookup (Centralised (CLU)/Distributed (DLU))*: A centralised lookup mechanism is effective for small-scale transactional processing; however, large-scale parallel executions of real-time application requests need a scalable and decentralised approach that provides consistent information and also reduces the number of lookups required to find a tuple in a partitioned database.
- *Incremental repartitioning cycle (Static (STC)/Incremental (INC))*: The repartitioning process needs to be incremental in order to handle dynamic workload characteristics, sudden data spikes, and architectural changes in the underlying DDBMS. Static repartitioning approaches are primarily suitable at the initial stage of system development when the workload characteristics are not fully known in advance. However, once the system has been up and running for an extended period of time, it is necessary to adopt an incremental approach to dynamically manage workload-related changes during daily operating periods.
- *Incremental repartitioning schedule (Implicit (IMP)/Explicit (EXP))*: The repartitioning scheme must provide mechanisms to administratively (i.e. explicitly) or automatically (i.e., implicitly) detect workload changes and measure the potential database metrics to trigger a repartitioning cycle. It also involves whether to perform on-/off-line data migrations immediately or wait for system idle times.

Table 2.6 compares twenty existing repartitioning schemes based on their strengths and weaknesses, in light of the abovementioned conceptual model of workload-aware incremental repartitioning. The abbreviations used in Table 2.6 are based on the acronyms presented (inside the rounded brackets) while describing different aspects of the proposed framework.

Table 2.6: A comparison of existing repartitioning schemes for shared-nothing OLTP systems

		Repartitioning aspects								
		Design	Workload-aware	Workload representation	Repartitioning strategy	Repartitioning cycle	Repartitioning schedule	Data migration	Data lookup	Optimisation
Repartitioning models	Schism [101, 102, 133]	DIN	✓	GR	EXP	STC	EXP	DM-OFF	CLU	DT+LBS
	MESA [98]	DIN	✓	–	–	STC	EXP	–	–	–
	Horticulture [16, 134]	SIN	✓	–	EXP	STC	IMP	DM-OFF	CLU	LBS
	SPAR [115]	SIN	✓	HR	EXP	INC	IMP	DM-OFF	CLU	DT
	HOPE [119–121]	SIN	✓	HR	EXP	STC	IMP	DM-OFF	CLU	DT+LBS
	Dtxn [103]	DIN	✓	–	EXP	STC	IMP	on-line	CLU	DT
	Elasca [123]	SIN	✓	–	EXP	STC	EXP	DM-OFF	CLU	DT+LBS
	ElasTras [107]	DIN	✓	–	EXP	INC	IMP	on-line	CLU	DT+LBS
	Wind [122]	SIN	✓	GR	EXP	STC	EXP	DM-OFF	CLU	DT
	Lute [92]	SIN	✓	GR	EXP	STC	EXP	DM-OFF	CLU	DT
	Squal [110, 111]	TRS	✓	–	EXP	INC	IMP	on-line	CLU	DT+LBS
	SWORD [93]	TRS	✓	CHR	EXP	INC	IMP	DM-OFF	CLU	DT+LBS
	SOAP [112, 113]	SIN	✓	–	EXP	STC	EXP	DM-OFF	CLU	DT
	H-Store [68, 135]	TRS	–	–	IMP	INC	IMP	DM-OFF	CLU	LBP
	E-Store [126]	TRS	✓	–	IMP	INC	IMP	DM-ON	CLU	DT+LBP+DM
	Accordion [130]	TRS	✓	–	IMP	INC	IMP	on-line	CLU	DT+LBP
	JetPants [51]	TRS	–	–	IMP	INC	IMP	DM-OFF	CLU	LBP+DM
	Vitess [3]	TRS	–	–	IMP	INC	IMP	DM-OFF	CLU	LBP+DM
	Gizzard [52]	TRS	–	–	IMP	INC	IMP	DM-OFF	CLU	LBP+DM
	MySQL Fabric [132]	TRS	–	–	IMP	INC	IMP	DM-OFF	CLU	LBP

2.8 Conclusions

Designing a scalable, interactive, and multi-tier Web application requires a high level of understanding of the life-cycle management of a shared-nothing distributed OLTP system, as well as the dynamics of workload characteristics. In this chapter, the architecture, deployment, transactional processing, issues and challenges in a distributed computing environment for OLTP DDBMS are discussed in detail. Furthermore, two primary data management architectures – replication and partitioning and architectures and issues related to multi-tier OLTP applications for achieving high scalability in the Web are also discussed. This elaborate discussion highlights the significance and necessity of workload-aware incremental database repartitioning, in particular, within a shared-nothing DDBMS, for serving real-time and interactive Web applications. Finally, a number of existing repartitioning schemes from the literature are presented, which leads in constructing a conceptual framework for the workload-aware incremental repartitioning process. In the following chapters, we will develop several incremental repartitioning schemes by covering different aspects of the aboved-mentioned conceptual framework, and filling out the gaps remaining in the existing literature, as highlighted in Table 2.6. In next chapter, we discuss and present the architecture system of the proposed workload-aware incremental repartitioning framework. This includes a novel transaction generation model capable of controlling transactional repetitions and unique transaction generations. Finally, a set of repartitioning KPIs is presented with their appropriate physical interpretations which are necessary to observe the characteristics of a repartitioning process.

Chapter 3

System Architecture, Workload Modelling, and Performance Metrics

In the previous chapter, the background, architectures, and challenges of distributed OLTP databases are discussed in detail, as well as data repartitioning techniques, along with the existing approaches from the literatures to tackle many of the impending challenges. Furthermore, a conceptual framework for a workload-aware incremental repartitioning process is also presented focusing on the design, granularity, performance, and management aspects. In this chapter, the proposed conceptual framework is elaborated with details on implementing an end-to-end demand-driven incremental repartitioning scheme, developing workload models for simulation based studies, and defining key performance metrics (KPIs) for evaluation purposes.

3.1 Introduction

In the previous chapter, a conceptual framework for a workload-aware incremental repartitioning scheme is outlined to overcome the shortcomings of existing approaches. In this chapter, a novel incremental database repartitioning framework is introduced, along with detailed problem formulation and two specific algorithmic strategies. The introduction of the unique distributed data lookup scheme, based on the concept of data *roaming*, enables the use of both *range-* and *consistent-hash* based initial data partitioning techniques. The simulation based evaluation process is discussed in depth where a distributed OLTP database simulation toolkit, named *OLTPDBSim* is introduced. *OLTPDBSim* uses the novel transaction generation model that is capable of restricting a target number of unique transaction generation. Later, three KPIs are introduced that measure 1) the impacts of DTs, 2) the server-level load-balance, and 3)

the cost of inter-server data migrations. The capabilities of these metrics are discussed in detail with appropriate physical interpretations to relate the measures to real-world scenarios. Overall, the contributions in detailing the functional building blocks of the proposed incremental repartitioning framework will enable the readers to comprehend the claims of our proposed repartitioning schemes in the following chapters.

The rest of the chapter is organised as follows. Section 3.2 presents a high-level system architecture of the proposed incremental repartitioning framework. Section 3.3 formally defines the repartitioning problem using mathematical notations and also discusses the evaluation process in a simulated environment. Section 3.4 details the characteristics of two particular OLTP workloads—*TPC-C* and *Twitter*, and presents a novel transaction generation model for simulation purposes as well. Section 3.5 discusses three unique KPIs in great detail in order to evaluate the quality of the proposed incremental repartitioning framework. Finally, Section 3.6 summarises the chapter.

3.2 System Architecture

An overview of the proposed workload-aware incremental framework, shallowly-integrated with a shared-nothing distributed OLTP system, is presented in Figure 3.1. Following the high-level overview of an OLTP system shown in Figure 2.1, four primary components are defined in a workload-aware repartitioning approach that work along side one another during the entire process, as outlined in Section 3.2.1.

3.2.1 Workload Processing in the Application Server

Incoming *client requests* for application data are distributed at the AS level. Modern applications use an elastic load-balancer to dynamically distribute end-users' requests based on AS-level resource consumptions, application-level latency criteria, and geographical locations. The *Workload Balancer* component in each AS further balances the transactional workload to handle simultaneous TCP connections asynchronously with the client [136]. Load-balance operations are further extended to store and distribute cache data objects (e.g., using *memcached* [137], *Redis* [138], etc.) and to set appropriate cookie information at the client-end.

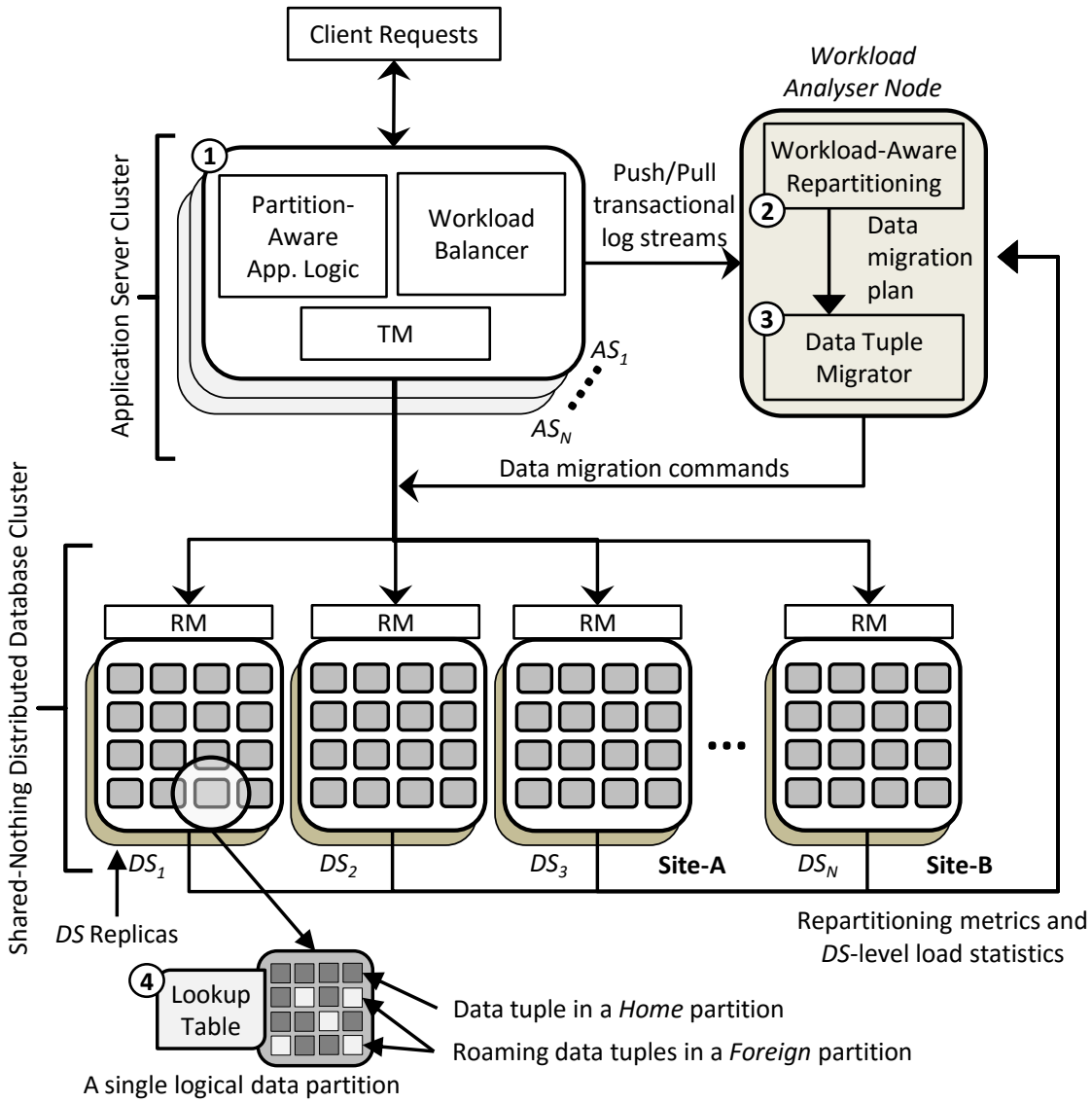


Figure 3.1: A high-level architecture of the shared-nothing distributed OLTP systems with workload-aware incremental repartitioning

To handle the underlying database partitioning schemes, partition-aware application logics are implemented using off-the-shelf libraries like *Oracle EclipseLink* [139] or *Apache OpenJPA* [140]. These libraries provide support for accessing *range-* and *hash-partitioned* data in an RDBMS, and APIs for the developers to utilise partitioning information. Finally, *TM* in each AS schedules, coordinates, and executes the transactional operations i.e., managing DTs issued by the application codes. Apparently, the transactional workload processing in the AS layer is entirely isolated from the incre-

mental repartitioning process, and neither has any undue impact on the other.

3.2.2 Workload Analysis and Repartitioning Decision

Streams of transactional logs are either continuously *pulled* by the *Workload Analyser Node* or periodically *pushed* by the AS. Transactions are then pre-processed for analysis either in a *time-* or *workload*-sensitive window. The *Analyser Node* can also cache the most frequently appeared tuple locations in a workload-specific catalogue file, which is kept updated on inter-partition data migrations. Upon triggering the *incremental repartitioning* scheme, the *Analyser Node* starts processing the transactional logs to generate an appropriate *repartitioning decision*.

3.2.3 Live Data Migration

Based on the output repartitioning decision, corresponding *data migration plans* are issued, which are then executed by the *Data Tuple Migrator* component. To execute the *data migration plan* without interrupting the ongoing transactional operations and providing an inconsistent data lookup result, the migration operations are carried out on-the-fly to migrate tuples between the logical data partitions within and across the DSs. At first, the migrating tuple is replicated to the target partition, and then the local data lookup tables residing in the source and destination partitions are updated in a single step to prevent any data inconsistency.

3.2.4 Distributed Data Lookup

As discussed in Chapter 2, any centralised data lookup mechanism is always a bottleneck achieving high availability as well as scalability requirements. In this thesis, we propose a sophisticated distributed data lookup that distributes the data tuple lookup tasks at the individual database partition level. Thus, data migration operations are totally transparent to DT processing and coordination. We adopt the concept of *roaming* [141] used in mobile telecommunication networks, where the location of a *roaming* mobile subscriber is kept updated to two different *Home Location Registers (HLRs)* in local and foreign networks. When a subscriber moves from one operator's network to

a foreign operator's network, then the mobile user is authenticated and authorised to use telecommunication services under this new network. If the foreign operator has an active roaming agreement with the local operator, then the subscriber is allowed to access the voice, data, and messaging services in the foreign network. Once the *roaming* user is whitelisted, the addresses of the network servers that are currently serving this user are stored and kept updated in the foreign HLR. Now, for example, if another mobile subscriber from the local network wants to make a voice call to this *roaming* user, then the local *Mobile Switching Centre (MSC)* server first looks up the subscriber location information in its cache, and, if not found, it then sends a request to the local operator's HLR. When the local HLR notices this lookup request is designated for a user that is currently *roaming* to a foreign network, it then forwards the address of the foreign HLR to the local MSC. The local MSC then directly communicates with the foreign HLR to retrieve the address of the MSC server serving the *roaming* user. Once the communication link between the local and foreign MSCs is established, the *roaming* user is paged to answer the incoming voice call. A similar technique is used to implement *roaming* in *Mobile IPv6* [142] networks with the aid of *care-of address (CoA)* [143].

We adopted the abovementioned approach for implementing a distributed data lookup in the proposed workload-aware incremental repartitioning architecture. At present, distributed databases support data lookup operations either in a centralised way in the *range*, or in a distributed way in the *hash/consistent-hash partitioned* [2] databases. To adopt on-the-fly physical data migrations for the repartitioning process, the data lookup technique in the *range partitioned* databases needs to be scalable and consistent enough to handle simultaneous updates in the lookup table, which is hard to achieve in such centralised systems. On the other hand, data lookup (especially for random read queries) using *consistent-hash* keys is much more scalable in the *consistent-hash partitioned* databases. However, there is no inherent mechanism in the lookup process that handles data *roaming* between the partitions. Presently, most of the OLTP applications use either a *range partitioned* database, which handles *range* queries through its centralised lookup mechanism, or a *hash/consistent-hash partitioned* database that handles such queries by writing complex application level codes. Therefore, there is always a dilemma for application developers to choose a particular type of database for serving multiple transactional query patterns.

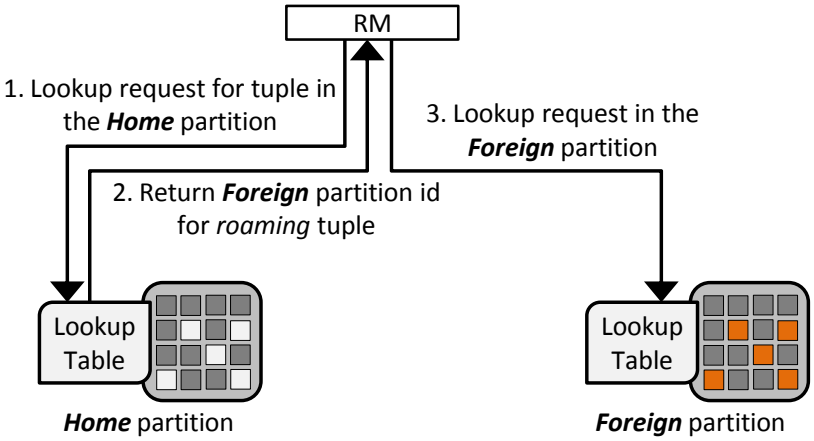


Figure 3.2: A distributed data lookup technique for *roaming* tuples with a maximum of two lookup operations in *home* and *foreign* data partitions

We propose a simple solution to this problem by introducing a *partition-level lookup table* to keep track of the *roaming* data while incrementally repartitioning the database to adopt workload changes. By maintaining a key-value list of *roaming* and *foreign* data ids with their corresponding partition ids, individual partitions can consistently answer the lookup queries. Tuples are assigned a permanent *home* partition id for their lifetime when the database is initially partitioned using a *range*, *hash*, or *consistent-hash* technique. The *Home* partition id only changes when a partition splits or merges, and these operations are overseen by the partition-aware application libraries (shown in Figure 3.1). Thus, scaling-out the database storage layer is independent of the distributed lookup scheme. As the tuple locations are managed by their *home* partitions, data inconsistency can be strictly prevented. Unless a tuple is fully migrated to another partition, and its *roaming* location is consistently written in the *home* partition's lookup table entry, the old partition continues to serve incoming data lookup requests. Once done, the old serving partition deletes the tuple from the storage.

When a tuple migrates to another partition within the process of incremental repartitioning, only its respective *home* partition needs to be aware of it. The target *roaming* partition treats this migrated tuple as a *foreign* tuple and updates its own lookup table accordingly. Meanwhile, the original *home* partition marks this tuple as *roaming* in its lookup table, and updates its current location with the *roaming* partition's id. As shown in Figure 3.2, the lookup process always sends a lookup request to a tuple's

home partition for its location information. If the tuple is not found in its *home* location, then the lookup table entry returns the most recent *roaming* location of the tuple, and redirects the lookup query to the *roaming* partition. Therefore, a maximum of two lookup operations are required to find a tuple within the entire database. Note that, through caching, an application can directly work with the *roaming* partition for any subsequent request for that tuple. If the tuple has been migrated again in the meantime, the direct request will fail, the cache is cleared, and a new request is made to the *home* partition as usual. With a high probability, individual data migration in the incremental repartition process requires updating the location information in the lookup tables of up to three RMs that are serving the *home* partition and *current* and *target roaming* partitions. Finally, this novel idea of a distributed data lookup, using the concept of *roaming*, enables the use of *consistent-hash* based initial data partitioning for OLTP databases that not only ensures high scalability but also reduces the burden on the application developers for large-scale distributed systems development in practice.

3.3 Workload-Aware Incremental Repartitioning

In this section, the problem of workload-aware incremental repartitioning, related properties, and relevant terms are formally defined.

3.3.1 Problem Formulation

Let $S = \{S_1, ..., S_n\}$ be the set of n *shared-nothing* physical DSs where each $S_i = \{\mathcal{P}_{i,1}, ..., \mathcal{P}_{i,m}\}$ denotes the set of m logical partitions residing in S_i. Let the set of data tuples that reside in $\mathcal{P}_{i,j}$ be denoted by $\mathcal{D}_{\mathcal{P}_{i,j}} = \{\delta_{i,j,1}, ..., \delta_{i,j,|\mathcal{D}_{P_{i,j}}|}\}$. Collectively, $\mathcal{D}_{S_i} = \bigcup_{\forall j} \mathcal{D}_{\mathcal{P}_{i,j}}$ and $\mathcal{D}_S = \bigcup_{\forall i} \mathcal{D}_{S_i}$ denote the set of data tuples in a DS S_i and the entire partitioned database, respectively, where $|\mathcal{D}_S|$ represents the data volume size of the entire database. For the sake of simplicity, the data volume size of the tuples is considered of equal unit size.

Let $\mathcal{W}$ be the workload observation time window consisting of $|\mathcal{W}| = \mathcal{R}\mathcal{W}$ transactions at the time of repartitioning where $\mathcal{R}$ is the transaction generation rate. Let the transactional workload in $\mathcal{W}$ be represented by $T = \{\tau_1, ..., \tau_{|T|}\}$ where τ_i's are the $|T|$

unique transactions observed in $\mathcal{W}$. Furthermore, the set of DTs and NDTs are respectively denoted as T_d and $T_{\hat{d}}$. Thus, $T = T_d \bigcup T_{\hat{d}}$ and $T_d \bigcap T_{\hat{d}} = \phi$ where $T_d = \{\tau_{d_1}, ..., \tau_{d_{|T_d|}}\}$ and $T_{\hat{d}} = \{\tau_{\hat{d}_1}, ..., \tau_{\hat{d}_{|T_{\hat{d}}|}}\}$. A DT τ_{d_i} and a NDT $\tau_{\hat{d}_i}$ can appear (i.e., repeat) multiple times within $\mathcal{W}$, hence their frequencies can be represented by $f(\tau_{d_i})$ and $f(\tau_{\hat{d}_i})$, respectively. Data tuples of a transaction τ may reside in one or more servers. Let $1 \leq s(\tau) \leq |S|$ denote the span of τ. For DTs, $s(\tau_{d_i}) > 1$ and for NDTs $s(\tau_{\hat{d}_i}) = 1$.

3.3.1.1 Problem Definition

In OLTP, as alluded to in Section 2.2.3 of Chapter 2, DTs create a major bottleneck for the system to reach eventual consistency due to the potential delays in arriving at consensus while executing a commit or abort operation. The end-users ultimately suffer from both potential inconsistency and the delays that are often collectively referred to as the impacts of DTs, I_d. Clearly, the span $s(\tau)$ of a transaction τ, defined as the number of servers where the data tuples of τ spread across, contributes directly to I_d. Collectively, the cost of a span is proportional to the average span of all DTs under consideration, i.e., $I_d \propto \overline{s(\tau_d)}$.

The impacts of DTs can be kept in check by ensuring that $\overline{s(\tau_d)} \ll |S|$. As the system has no control on how the end-users initiate a transaction, the only effective way to reduce I_d is by consolidating the span of DTs to fewer servers by physically moving some of their data tuples. Such reactive data migration plans may put the inter-server communication networks under stress, causing network delays that may lead to potential SLA violations. This is referred to as the indispensable data migration cost, D_m, associated with minimising the impacts of DTs. Data migration plans are expected to be triggered simultaneously for a large number of targeted DTs that take place in parallel so long as the sending- and receiving-end servers of concurrent data transfers remain mutually exclusive. Collectively, the cost of data migration is proportional to the average number of cycles needed to move all the data tuples targeted.

Data migration plans may also severely affect the overall balance of server-level loads when the data tuples are not eventually stored evenly across all servers, leading to potential increases in rate of hardware failure. In the long run, some of the

servers may be used significantly more than others and reach their life expectancy earlier, leading to unplanned interruptions of services. This is referred to as the cost of (disturbing) load balancing, L_b, which is proportional to the variance in data volume across the servers. We now define the key optimization problem addressed in this thesis as follows.

Main Problem: *How to minimise the impacts of DTs with minimal data migration while maintaining the load balancing of the servers.*

Sub-problem 1: *How to detect whether the impacts of DTs of a system has crossed its critical level and how often the system should be checked.*

Sub-problem 2: *How to trade-off the impacts of DTs and the associated costs of data migration and server-level load imbalancing.*

In order to address the problems adequately, we need to first develop specifications and means to estimate I_d, D_m, and L_b accurately. These estimates should also be normalised appropriately so that they are not affected by the high scalability of OLTP systems. The first part of **Sub-problem 1** may then be deemed as straightforward: namely, setting up some user-defined thresholds on these metrics after considering the workload models and database schemas. The second part of **Sub-problem 1** involves investigating both proactive and reactive approaches for triggering incremental repartitioning. We plan to address **Sub-problem 2** using both graph-theoretic and procedural approaches, focusing on computational efficiency and lower-bounds. Successful implementation of incremental repartitioning is dependent on designing an efficient data lookup mechanism, which is transparent to the end-users from the data migration plans. This chapter has already presented a novel data lookup mechanism from that perspective in Section 3.2.4.

3.3.2 Incremental Repartitioning

In this thesis, three strategies for incremental repartitioning are used. The first strategy takes the transactional workload as an input and represents it as a workload network (e.g., graph or hypergraph). The workload representations are then clustered using graph-cut libraries that minimise the edge cuts. This eventually reduces the number of DTs in the workload while the output clusters are distributed accordingly within

the logical data partitions in order to achieve load-balance across the DSs. In short, the graph-cut based strategy is an implicit approach for repartitioning the database where minimisation of DTs and load-balance come as the by-products. However, this approach does not guarantee minimum data migrations during the repartitioning process and there is no explicit control over how to maximise the other objectives of minimising the overall impacts of DTs in the system and load-balance.

The second approach, on the other hand, operates exhaustively over the given transactional workload and uses a specific heuristic either to maximise a specific repartitioning objective or to balance multiple objectives by minimising their cost of per data migration. This explicit repartitioning strategy is particularly useful for situations when a system requires to achieve any particular repartitioning goal while performing minimum physical data migrations over the network.

Finally, in the third approach, a frequent tupleset mining technique is used to reduce the workload network size for making on-demand and more frequent repartitioning decisions. Furthermore, both graph theoretic and greedy heuristic approaches are combined in this latter approach, where graph a min-cut technique is used to cluster compact workload networks created through transactional mining, and transactional associativity based ranking is used to determine the best data migration plan for on-demand incremental repartitioning.

3.3.3 Evaluation Process

The performance of the proposed incremental repartitioning schemes is evaluated using simulated shared-nothing distributed RDBMS environments under synthetic OLTP workloads. A novel simulation platform called *OLTPDBSim* is designed and developed from scratch using the *Stochastic Simulation in Java (SSJ)* [144] library. A unique transaction generation model is developed and incorporated with *OLTPDBSim* for simulating different variants of transactional workloads. Using a parametric representation of the workload, underlying database, and the system under test, a simulated shared-nothing OLTP environment is created. It contains S physical servers, and an initial database where the tables are partitioned into $\mathcal{P}$ logical partitions using either

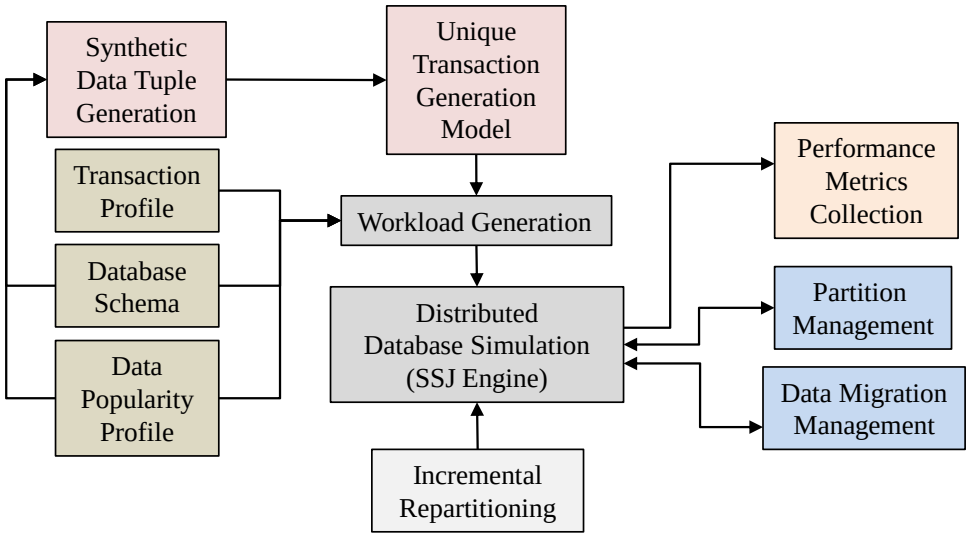


Figure 3.3: An overview of the *OLTPDBSim* simulation framework and its components

range or *consistent-hash* based initial data partitioning. Figure 3.3 presents a high-level overview of the *OLTPDBSim* simulation framework.

3.4 Workload Characteristics and Transaction Generation

In this section, the characteristics and specific implementations of two mainstream OLTP workloads – *TPC-C* and *Twitter* – are presented. Later, a novel transaction generation model is proposed with a detailed sensitivity analysis, along with specific reviews of the existing literature on synthetic workload generation.

3.4.1 Workload and Transaction Profile

One of the major components in *OLTPDBSim* is the OLTP workload generation process that involves synthetic data tuple generation based on a data popularity profile and database schema. This is followed by workload generation of specific types based on their transaction profiles, and controlled by the transaction generation model. To evaluate the performance of various incremental repartitioning strategies, two specific types of OLTP workloads are developed following their individual schema, guidelines, and specifications for transaction profile and mix. A brief description of each of the workloads is presented in the following subsections.

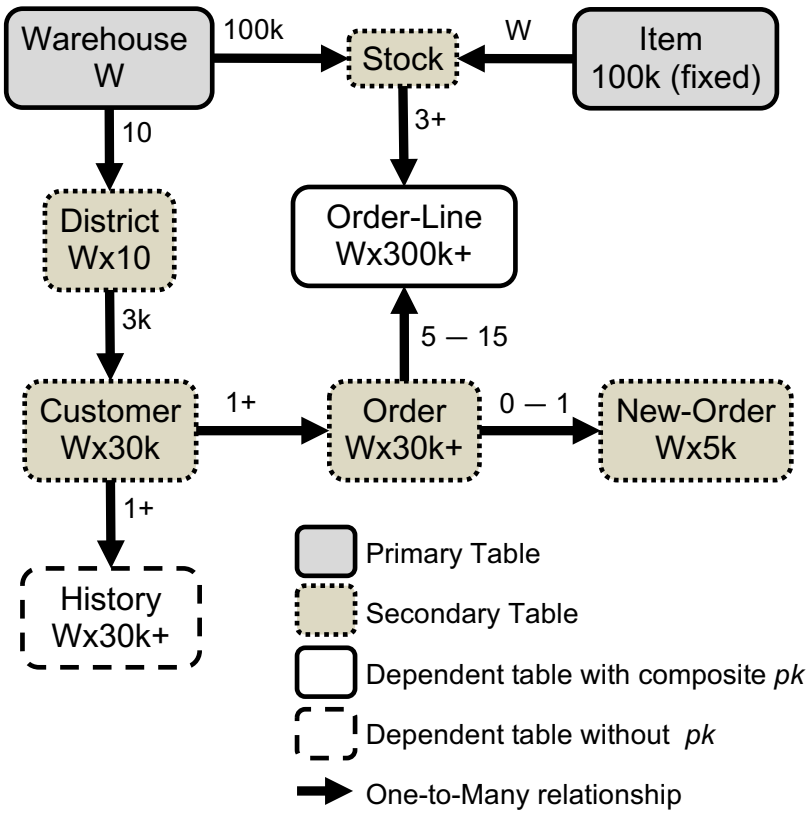


Figure 3.4: The schema diagram of the *TPC-C* OLTP database

3.4.1.1 TPC-C Workload

TPC-C [145] is the industry standard workload specification for evaluating OLTP database performance. A typical *TPC-C* database contains nine tables, and five transaction classes, and it simulates an order-processing transactional system within geographically distributed districts and associated warehouses. The *TPC-C* schema diagram is shown in Figure 3.4. It is a write-heavy benchmark where transactions are concurrent, and complex, and they often perform multiple table *JOIN* operations. Among the nine tables, 'Stock' and 'Order-Line' tables grow faster in volume, thus all the logical database partitions are not homogeneous in size if the database is initially *range partitioned*. New tuples are also inserted into 'Order' and 'Order-Line' tables using the 'New-Order Transaction' which usually occupies nearly 44.5% of the workload. The table partitions are distributed equally within the physical servers following a round-robin placement strategy, thereby ensuring equal load distribution occurs at the initial

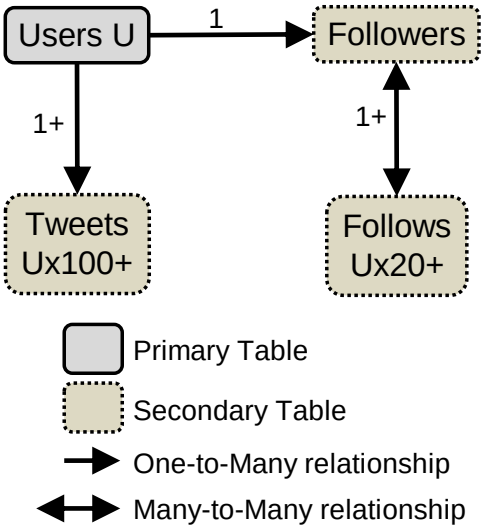


Figure 3.5: The schema diagram of the *Twitter* OLTP database

stage of the database's life-cycle. The five transaction classes are weighted from *heavy* to *light* (in terms of processing), and these classes appear at *high* to *low* frequencies within the workload. Two of the most high frequency transaction classes in *TPC-C* have strict response time requirements and cover almost 87.6% of the total generated transactions. The synthetic data generation process shown in Figure 3.3 follows Zipf's law [118] for generating 'Warehouse' and 'Item' tables' data, and it uses the table relationships to generate others. In overall, *TPC-C* specification guidelines are followed closely while implementing the workload for *OLTPDBSim*.

3.4.1.2 Twitter Workload

The *Twitter* workload is based on the popular micro-blogging Web site and it represents the social-networking OLTP application domain. In *OLTPDBSim*, only a subset of the *Twitter* workload is implemented by closely following the implementations of the *Oltpbench* [146] workload generator. It primarily includes the 'User'–'Tweets' and 'Follower'–'Followee' relationships. The workload contains four tables and five classes of transactions without having any *JOIN* operations. The schema diagram is shown in Figure 3.5. Among the five classes of transactions, the most frequently occurring transactions are selecting tweets for a given user id, and selecting tweets from users

that a given user follows, and that altogether cover almost 98.9% of the entire workload. Although the implemented *Twitter* workload is not an actual representative of the workload driven at the micro-blogging site, it sufficiently captures the complex social graph network, which is heavily skewed with many-to-many 'Follows'–'Follower' relationships following Zipf's law [118].

3.4.2 Modelling Transactional Workload

In this section, some of the key existing techniques for modelling synthetic transactional workloads are discussed. Later, a novel transaction generation model for synthetic OLTP workload generation is presented with its mathematical properties, followed by a detailed sensitivity analysis.

3.4.2.1 Existing Transactional Workload Generation Models

The primary goal of most of the existing session-based [147, 148] synthetic workload generation tools is to measure the back-end performance in terms of throughput achieved from parallel user loads or scalability testing by simultaneously increasing the number of users until the system saturates. *Cloudstone* [149] was among the earliest to provide a micro-benchmark toolkit for analysing the performance of Web 2.0 applications. It uses an open-source social-events Web application called *Olio* for benchmarking purpose, and includes a *Markov* process based transactional workload generator called *Faban* [150].[1] It is an open-source synthetic workload generator which aims to facilitate performance and user load testings for Web applications. It allows creating simple to complex workloads by specifying different mixes of transaction classes for generating new transactions. Furthermore, inter-transaction generation in *Faban* follows standard distributions like negative exponential, uniform, and fixed time. The number of simulated users can also be varied during the runtime. Nonetheless, *Faban* provides various options and configuration parameters to generate different workload scenarios at the transaction class level while simultaneously increasing the number of system users. However, it does not include any model for controlling the repetition of

[1] http://faban.org

transactions over time, which is a rather common case in OLTP workloads. *CloudSuite 2.0* [151, 152], which is a more recent and compact development based on both *CloudStone* and *Faban*, provides updated documentations for installation, configuration, and integration with different components.

The *Rain* Workload Generation Toolkit [153, 154] is yet another workload generation model based on the statistical properties of the OLTP applications. It supports workload variations in three key directions—system load and resource consumption, transactional and operational mixes, and data popularity and transient hot spots. Based on the changes in statistical properties and probability distributions, *Rain* provides a way of performing effective cost-benefit analysis to make 'what-if' scenario-based decisions. *Rain* also uses *Olio* target Web application and *CloudStone* project as a benchmark suite. More recently, *OLTP-Bench* [146, 155, 156] was developed primarily to benchmark JDBC-enabled RDBMS with various OLTP workloads. It supports three types of transaction generation: 1) closed-loop, 2) open-loop, and 3) semi-open-loop. With the closed-loop setting, the generator initialises a fixed number of worker threads that repeatedly generate new transactions with a random think time. With the open-loop setting, new transaction generation by the workers follows a stochastic process. Finally, in the semi-open-loop setting, the worker threads pause for a random think time before generating a new transaction following the open-loop model. Although both *Rain* and *OLTP-Bench* ensure good statistical properties for OLTP workload generation, none of them support unique transaction generation with controlled repetition to automatically generate different kinds OLTP workload in a parameterised way.

3.4.2.2 Proposed Transaction Generation Model

In order to evaluate the performance of the proposed workload-aware incremental repartitioning schemes, one needs to be careful to avoid any undue influences from any external entity (e.g., graph clustering libraries). The easiest way to achieve this is by designing a transaction generation model capable of generating transactions at a target rate $\mathcal{R}$ while maintaining a target unique transaction proportion $\mathcal{U} = |T|/|\mathcal{W}|$ in the observation window $\mathcal{W}$. This will allow us to use similar sized workload networks while comparing the performance of the same repartitioning algorithm on different

scenarios, or different repartitioning algorithms under the same scenario. The transaction generation model, however, has to be flexible enough to allow us to control the transaction mix such that at any transaction generation instance, a new transaction is introduced with probability p; otherwise an existing transaction is repeated. In contrast, real-life OLTP workloads do not necessarily maintain a fixed target percentage of unique transaction generations. Therefore, by setting the system parameters $\langle \mathcal{R}, p, \mathcal{W}, \mathcal{U} \rangle$ with appropriate values, a wide variety of transactional systems can be modelled for simulation purposes. Let $\eta = \mathcal{R}p$ be the average generation rate of new transactions in the current observation window. Obviously $\mathcal{U}$ is bounded as follows:

$$\eta \leq \mathcal{U} \leq \mathcal{R}. \tag{3.1}$$

Implementation of this transaction model is quite straightforward. The only challenge remaining is to set a limit on how far back the generator should look for transaction repetitions so that the target $\mathcal{U}$ is achieved. Let the generator consider only the latest $\mathcal{U}'|\mathcal{W}|$ unique transactions and randomly select one of them with uniform distribution for repetition. For average-case analysis, $\mathcal{U}'$ may now be restricted within the following lower and upper bounds

$$\mathcal{U} - \eta \leq \mathcal{U}' \leq \mathcal{U}. \tag{3.2}$$

If $\mathcal{U}' < \mathcal{U} - \eta$, the target $\mathcal{U}$ is likely to be under-achieved as the number of unique transactions in the repetition pool is less than $(\mathcal{U} - \eta)|\mathcal{W}|$. On the other hand, if $\mathcal{U}' > \mathcal{U}$, the target $\mathcal{U}$ is likely to be over-achieved as the number of unique transactions in the repetition pool is more than $\eta|\mathcal{W}|$.

The functional model

$$\mathcal{U}' = \mathcal{U}\left(1 - \left(\frac{\eta}{\mathcal{U}}\right)^q\right) \tag{3.3}$$

can ensure the abovementioned boundaries when q is finite and $q > 1$. Our empirical analysis has found that the target $\mathcal{U}$ can be achieved best for $q = 2$.

The proportion of unique transactions in the generated transactional workload will vary from the target $\mathcal{U}$ if the observation window size deviates from $\mathcal{W}$ used by the

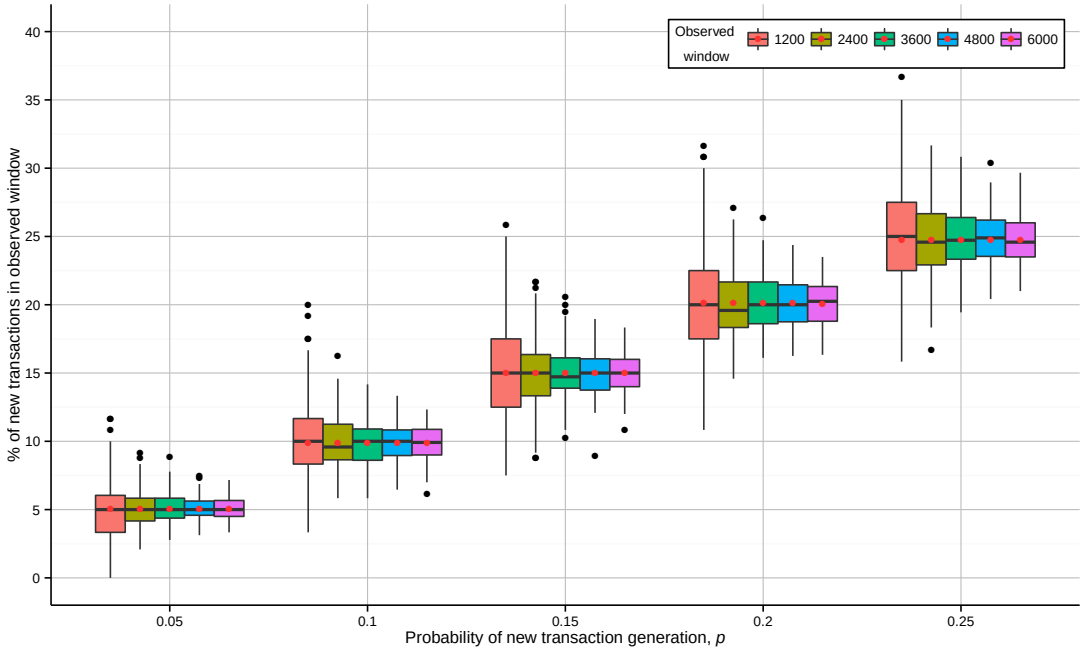


Figure 3.6: Target percentage of new transaction generations using different observed windows with $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle = \langle 1, 25\%, \in \{1200, 2400, 3600, 4800, 6000\}, \in \{0.05, 0.1, 0.15, 0.2, 0.25\} \rangle$

transaction generation model. If the window size is smaller than $\mathcal{W}$, the proportion will be higher for $p < \mathcal{U}$ and lower for $p > \mathcal{U}$. The opposite occurs when the window size is larger than $\mathcal{W}$.

3.4.2.3 Sensitivity Analysis

A detailed sensitivity analysis of the proposed transaction generation model is performed for different combinations of $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle$ by varying p. Given $\mathcal{R}$ of 1 transaction per second (tps), the target is set to generate 25% unique transactions within an observation time window $\mathcal{W}$ of 1 h (3600 s)) that will contain $|\mathcal{W}| = 3600$ preceding transactions. This generation process lasts for a total period of 24h. The value of p is varied from 0.05 to 0.25 with a step size of 0.05. Appendix A.1 lists the MATLAB code written to perform the sensitivity analysis of the proposed transaction generation model. Figure 3.6 presents the consistent generation of a target percentage of new transactions with respect to different values of p, observed over five observation windows $\mathcal{W}$ = 1200, 2400, 3600, 4800, and 6000 seconds. This is an essential property of any transaction generator, and hence it is observed and satisfied for the proposed

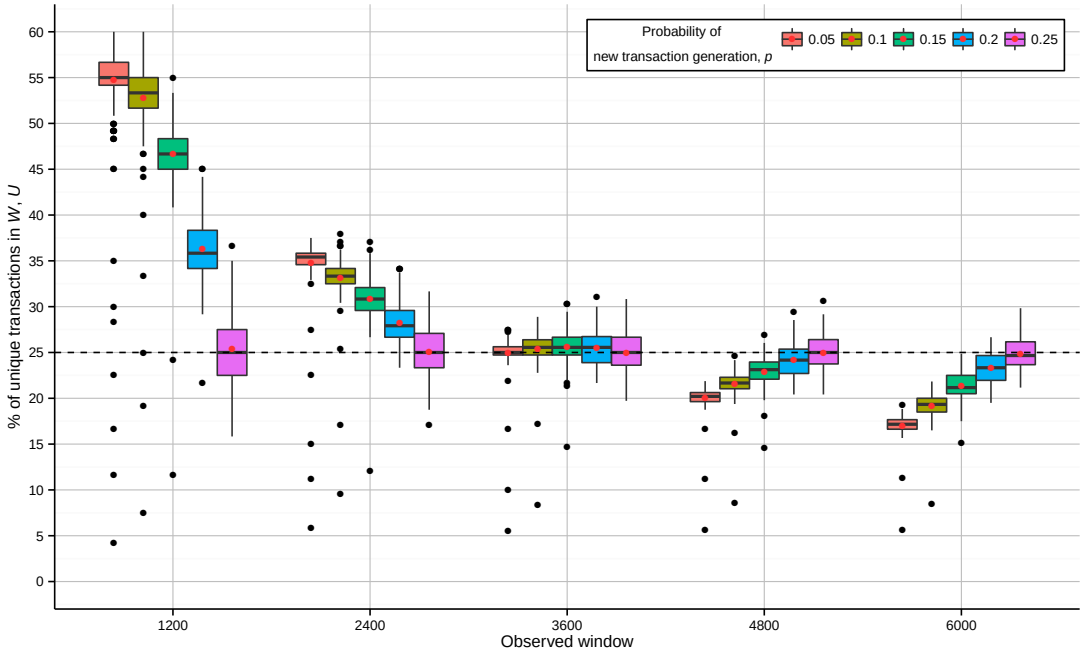


Figure 3.7: Unique transaction generations using a restricted repetition pool with $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle = \langle 1, 25\%, 3600, \in \{0.05, 0.1, 0.15, 0.2, 0.25\} \rangle$

model as well, for any given p across all observation windows. The whiskers and outliers observed in the plots are due to the data points from the initial warming up phase of the model, and these can be ignored.

Figure 3.7 shows the consistency of the proposed transaction generation model for generating a target of 25% unique transactions over a particular window size of 3600 s using a restricted repetition pool based on (3.2) and (3.3). As one can observe, the proposed model has met the target, observed over 3600 s for any values of p. Furthermore, as the observation window size decreases from 3600 s down to 1200 s and 2400 s, the percentage of unique transactions increases for cases $p < \mathcal{U}$. On the other hand, when the observed window size increases, the proportion of $\mathcal{U}$ in $\mathcal{W}$ decreases for cases $p < \mathcal{U}$. In both of these cases, the target is met when $p = \mathcal{U}$, as per (3.1). Furthermore, for observed windows 4800 s and 6000 s, the size of the repetition pools (e.g., 644 and 500, respectively, for $p = 0.1$) is less than the required $(\mathcal{U} - \eta)|\mathcal{W}|$ amount of 720 and 900 transactions, respectively. On the other hand, for window sizes 1200 s and 2400 s, the target is overachieved as the size of the repetition pools (e.g., 884 and 836, respectively, for $p = 0.1$) is much larger than the required $\eta|\mathcal{W}|$ amount of 120

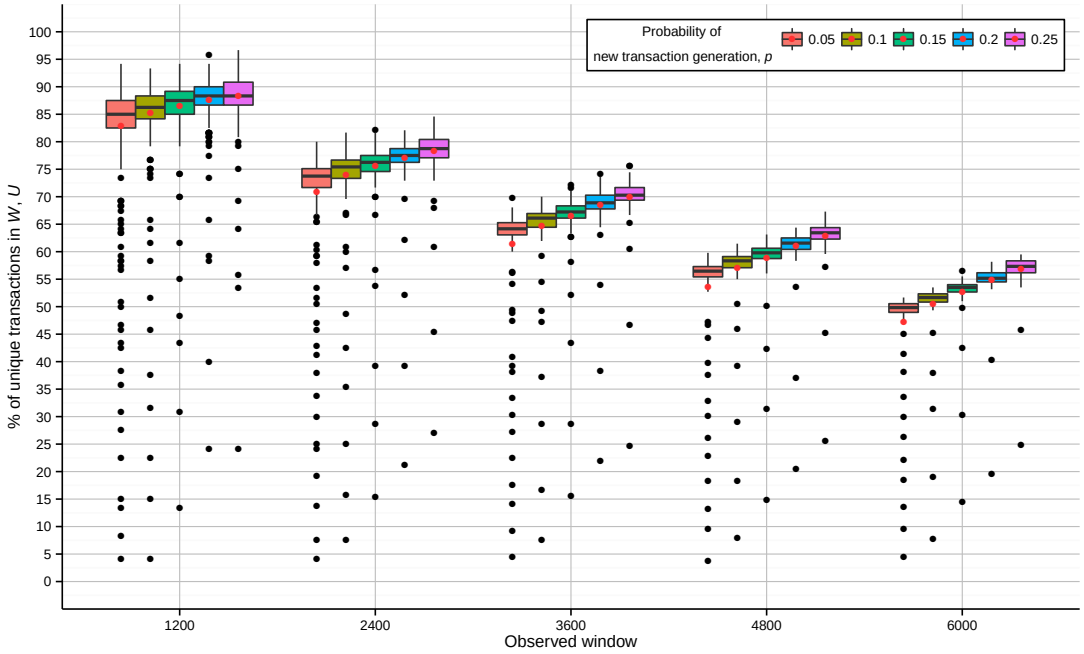


Figure 3.8: Unique transaction generations using an unrestricted repetition pool with $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle = \langle 1, \in \phi, 3600, \in \{0.05, 0.1, 0.15, 0.2, 0.25\} \rangle$

and 240 transactions, respectively.

Figure 3.8, shows the proposed transaction generation model's capability of generating unique transaction mixes for an observation window size of 3600 s using an unrestricted repetition pool. As can be seen, as the observed window size increases there are fewer unique transactions per observation due to a smaller repetition pool, and the opposite occurs when the window size decreases.

Table 3.1: The repetition statistics for $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle = \langle 1, 0.25, 3600, \in \{0.05, 0.1, 0.15, 0.2, 0.25\} \rangle$ using a restricted repetition pool

p	*0.05*	*0.10*	*0.15*	*0.20*	*0.25*
Number of unique transactions used for repetition	864	756	580	320	1
Mean inter-repetition interval	778 s	653 s	498 s	272 s	1 s
Mean frequency of repetitions per unique transaction	4.62	5.51	7.24	13.21	3600

Table 3.2: The repetition statistics for $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle = \langle 1, \in \phi, 3600, \in \{0.05, 0.1, 0.15, 0.2, 0.25\} \rangle$ using an unrestricted repetition pool

p	*0.05*	*0.10*	*0.15*	*0.20*	*0.25*
Number of unique transactions used for repetition	3600	3600	3600	3600	3600
Mean inter-repetition interval	3186 s	3072 s	3012 s	2997 s	3009 s
Mean frequency of repetitions per unique transaction	1.13	1.17	1.19	1.20	1.19

Table 3.1 lists the statistics of transactional repetitions using a restricted repetition pool for generating 25% unique transactions within the observation window of 3600 s for different values of p, given $\mathcal{R} = 1$. For smaller values of p, the repetition pool tends to be larger, therefore each unique transaction is picked up less often (less frequently with a longer inter-repetition interval) for repetitions. The opposite is true for larger values of p. Finally, the repetition frequencies get restricted at 1, when the η and $\mathcal{U}$ become equal, following the bounds mentioned in (3.1).

Table 3.2, on the other hand, lists the repetition statistics for unique transaction generations within 3600 s of the observation window using an unrestricted repetition pool. In this mode of generation, an individual unique transaction gets very limited opportunity to reappear in the workload due to a larger repetition pool size equal to $|\mathcal{W}|$. This phenomenon can be observed in Figure 3.8 as well. Overall, the proposed model shows its capability in controlling transactional repetitions in such a way that it can meet the goal of generating a target percentage of unique transactions observed over a given time window. This capability is particularly important in restricting workload network size while comparing different repartitioning schemes by avoiding any undue external influence.

3.5 Repartitioning Performance Metrics

In this section, several workload-aware incremental database repartitioning KPIs are proposed and analysed with their physical interpretations. Traditionally, system administrators measure the scalability of a system under stress primarily in terms of throughput. The *Universal Scalability Law (USL)* [157] is one well known measure for scaling of such systems. The law combines 1) the effects of achieving equal system throughputs for the given load, 2) the cost of shared resources, 3) depreciating returns from contention, and 4) negative returns from incoherency. In a simplistic way, by using a 3-way measure of the level of concurrency, contention, and coherence, the USL provides a way of calculating the relative scalability of a software or hardware platform.

Based on the same arguments, the repartitioning performance should be measured in a way that also reflects the scalability measures of a distributed OLTP database. In

evaluating the performance of the incremental repartitioning, previous work [93, 101] only measures the percentage reductions in DTs within the entire workload. However, this single measure fails to provide any meaningful insights into how the impacts of DTs are minimised. Furthermore, there are no explicit measures available for overall load-balance and data migrations during an incremental repartitioning process.

Here, three unique KPIs are proposed to measure the successive repartitioning quality achieving three distinctive objectives to minimise 1) the impacts of DTs, 2) server-level load-imbalance, and 3) number of inter-server physical data migrations. The first metric measures the impact associated the frequency of DTs and their related spanning cost that are directly related to system resource consumptions. The second metric measures the tuple-level load distribution over a set of physical servers using the *coefficient of variation*, which effectively shows the dispersion of data tuples over successive periods of observation. The third metric measures the mean inter-server data migrations for successive repartitioning processes. This, in turn, is equal to the I/O parallelisms achieved while processing simultaneous data migration operations by the underlying OS, disk, and network cards. These KPIs are outlined in the following subsections.

3.5.1 Impacts of Distributed Transaction

Based on the definitions from Section 3.3, the cost of DT for any given τ_{d_i} is the product of its spanning cost $s(\tau_{d_i})$ and repetition frequency $f(\tau_{d_i})$, observed within $\mathcal{W}$. Here, $s(\tau_{d_i}) = |\{\forall v \in \tau : (server)a \mid \exists(server)a\ \exists(partition)b\ \exists(tuple)c\ \delta_{a,b,c} = v\}|$, which denotes the number of physical servers involved in processing τ_{d_i}; whereas, $s(\tau_{\hat{d}_i}) = 1$ for any NDTs. Note that, in reality, this cost represents the overhead of network I/O while processing the DTs. Equation (3.4) below defines the spanning cost of DTs within $\mathcal{W}$ for all $\tau_{d_i} \in T_d$

$$c(T_d) = \sum_{\forall \tau_{d_i} \in T_d} f(\tau_{d_i}) s(\tau_{d_i}). \tag{3.4}$$

Similarly, (3.5) denotes the spanning cost of NDTs for all $\tau_{\hat{d}_i} \in T_{\hat{d}}$

$$c(T_{\hat{d}}) = \sum_{\forall \tau_{\hat{d}_i} \in T_{\hat{d}}} f(\tau_{\hat{d}_i}). \tag{3.5}$$

Finally, the impacts of DTs is defined as:

$$I_d(\mathcal{W}) = \frac{c(T_d)}{c(T_d) + c(T_{\hat{d}})} \tag{3.6}$$

According to the definition in (3.6), the impacts of DTs is estimated in real domain within the range $[0,1]$; the lower and upper limits are reached when there are no DTs ($T_d = \phi$) and no NDTs ($T_{\hat{d}} = \phi$), respectively. This impact metric, however, suffers from the following shortcomings:

(i) It is insensitive to the server spanning cost of the DTs when they significantly outnumber NDTs, i.e., $|T_d| \gg |T_{\hat{d}}|$. For example, when 90% of transactions are distributed, the impact metric $I_d(\mathcal{W})$ varies within a very narrow range of $[\frac{0.9\times2}{0.9\times2+0.1} \equiv 0.947, 1)$, where 2 is the average spanning cost of DTs.

(ii) It is unstable when observed in a sliding window as the frequency $f(\tau)$ of each transaction τ is estimated locally within the window, without considering the trend. For example, I_d in two reasonably overlapped windows with the same set of unique transactions may vary significantly due to frequency differences.

The first shortcoming is addressed by expressing the impact relative to the worst scenario, when every transaction is spanned across all the servers S. The second problem is mitigated by estimating the frequency of a transaction at the current instance from its expected period of recurrence, calculated from the exponential moving average of its so far observed periods.

Let $t_\tau(k)$ denote the time of the k-th occurrence of the transaction τ in the system from the start of the system. Its observed period $\Upsilon_\tau(k)$ and expected period of recurrence $\tilde{\Upsilon}_\tau(k)$ is at that instance updated as follows:

$$\Upsilon_\tau(k) = t_\tau(k) - t_\tau(k-1) \tag{3.7}$$

for $k > 1$ and

$$\tilde{\Upsilon}_\tau(k) = \begin{cases} \mathcal{U}', & k = 1; \\ \alpha\Upsilon_\tau(k) + (1-\alpha)\tilde{\Upsilon}_\tau(k-1), & k > 1 \end{cases} \tag{3.8}$$

where the exponential averaging coefficient α is a constant between 0 and 1; a higher α discounts older observations faster. The expected period of recurrence is initialised with the number of unique recurring (not new) transactions $\mathcal{U}'$, derived in (3.3) from the target number of unique transactions $\mathcal{U}$ and the average number of new transactions added in the observation window $\mathcal{W}$, under the uniform recurrence assumption.

Let $T_u = \{\tau_{u_1}, ..., \tau_{u_{|T_u|}}\}$ be the set of unique transactions in $\mathcal{W}$. Considering that frequency is reciprocal of period, the impacts of DTs in $\mathcal{W}$ may now be estimated with the following novel estimator

$$\begin{aligned} I_d(\mathcal{W}) &= \frac{\sum_{\forall j} s(\tau_{u_j}) f(\tau_{u_j})}{|S| \sum_{\forall j} f(\tau_{u_j})} \\ &= \frac{\sum_{\forall j} \left(\frac{s(\tau_{u_j})}{\Upsilon_{\tau_{u_j}}(k_j)} \right)}{\sum_{\forall j} \left(\frac{|S|}{\Upsilon_{\tau_{u_j}}(k_j)} \right)} \end{aligned} \tag{3.9}$$

where k_j denotes the number of occurrences of unique transaction τ_{u_j} from the start of the system. Note that, according to (3.9), $I_d(\mathcal{W})$ is bounded in the range $[\frac{1}{|S|}, 1]$; therefore, it is also an indicator of server-level resource usage due to a particular transactional workload.

3.5.2 Server-level load-balance

The measure of load-balance across the physical servers is determined from the growth of the data volume with the set of physical servers. If the standard deviation of data volume $\sigma_{\mathcal{D}_S}$ for all the physical servers is calculated, then the variation of distribution of tuples within the servers can be observed. The coefficient of variation (C_v) defines the ratio between $\sigma_{\mathcal{D}_S}$ and $\mu_{\mathcal{D}_S}$, the mean data volume per server, and it is independent of any unit of measurement. C_v can tell the variability of tuple distribution within the servers in relation to the mean data volume $\mu_{\mathcal{D}_S}$. Equation (3.10) below estimates the server-level load-balance of the entire database cluster, in terms of C_v, independent of any observation instance.

$$L_b = C_v = \frac{\sigma_{\mathcal{D}_S}}{\mu_{\mathcal{D}_S}} \tag{3.10}$$

where $\mu_{\mathcal{D}_S} = \frac{1}{|S|}\sum_{i=1}^{|S|}|\mathcal{D}_{S_i}|$ and $\sigma_{\mathcal{D}_S} = \sqrt{\frac{1}{|S|}\sum_{i=1}^{|S|}\left(|\mathcal{D}_{S_i}| - \mu_{\mathcal{D}_S}\right)^2}$.

However, C_v, as a measure of load-balance, does not guarantee a normalised value of L_b within a specific range. For instance, if $\mu_{\mathcal{D}_S}$ tends to zero, C_v approaches to infinity. On the other hand, if $|\mathcal{D}_{S_i}|^2 > 2\mu^2_{\mathcal{D}_S}$ for any S_i, then the value of L_b will be greater than 1. Thus, the measure of L_b in (3.10) does not have any upper limit. To bound the value of L_b in the range of 0 and 1, the worst-case scenario is considered, where all the data tuples reside in a single server and the remaining $(|S|-1)$ servers in the cluster are empty. Based on this assumption, the worst-case measure of C_v can be derived as follows:

$$\begin{aligned}
\sigma^2_{\mathcal{D}_S}\big|_{\max} &= \frac{1}{|S|}\left[\left(|\mathcal{D}_S| - \mu_{\mathcal{D}_S}\right)^2 + \left(|S|-1\right)\left(0-\mu_{\mathcal{D}_S}\right)^2\right] \\
&= \frac{1}{|S|}\left[\left(|\mathcal{D}_S| - \frac{|\mathcal{D}_S|}{|S|}\right)^2 + \left(|S|-1\right)\frac{|\mathcal{D}_S|^2}{|S|^2}\right] \\
&= \frac{|\mathcal{D}_S|^2}{|S|}\left[\frac{1}{|S|^2}\left(|S|-1\right)^2 + \frac{1}{|S|^2}\left(|S|-1\right)\right] \\
&= \frac{|\mathcal{D}_S|^2}{|S|^2}\left(|S|-1\right) \\
\therefore C_v\big|_{\max} &= \frac{\sigma_{\mathcal{D}_S}\big|_{\max}}{\mu_{\mathcal{D}_S}\big|_{\max}} = \sqrt{|S|-1}
\end{aligned} \tag{3.11}$$

Finally, by utilising (3.11), L_b can be normalised within the range of $[0,1]$ as

$$L_b = \frac{C_v}{C_v\big|_{\max}} = \frac{\sigma_{\mathcal{D}_S}}{\mu_{\mathcal{D}_S}\sqrt{|S|-1}} \tag{3.12}$$

3.5.3 Inter-Server Data Migrations

In any $\mathcal{W}$, the total of inter-server data migrations within a repartitioning process can be normalised by dividing it with the mean data volume $\mu_{\mathcal{D}_S}$. Let D_m be the inter-server data migration metric with respect to an observed $\mathcal{W}$ during a particular repartitioning cycle. Then, D_m may be estimated as

$$D_m = \frac{M_g}{\mu_{\mathcal{D}_S}} \tag{3.13}$$

where M_g is the total number of inter-server data migrations during $\mathcal{W}$.

However, if the number of servers increases or decreases in the cluster without changing the total data volume $|\mathcal{D}_S|$, the mean data volume $\mu_{\mathcal{D}_S}$ also decreases or increases, concomitantly. Therefore, this simple cost metric in (3.13) does not provide a normalised measure with appropriate physical interpretations. The cost of inter-server data migrations can be better estimated considering the parallelisms in OS and internetworking levels, while transferring data between a pair of DSs. Therefore, by estimating the total amount of pair-wise data migrations, and then dividing this number by the total data volume $|\mathcal{D}_S|$, the measure of D_m can be appropriately normalised.

Given $M = \{m_1, m_2, ..., m_{|S|(|S|-1)}\}$ as the sorted set of integers representing the number of pair-wise 1-way inter-server data migrations during a repartitioning cycle within $|S|$ servers, the total number of inter-server data migrations can be defined as

$$\widetilde{M_g} = \sum_{\forall i} m_i. \tag{3.14}$$

Now, the problem is to find the number of *mutually-exclusive* pairs of servers to represent the set of $\big(|S|(|S|-1)\big)$ pairs. This process involves $\frac{|S|(|S|-1)}{\lfloor \frac{|S|}{2} \rfloor}$ or $2(|S|-1)$ iterations to complete where $\lfloor \frac{|S|}{2} \rfloor$ *mutually-exclusive* server pairs exist. Therefore, to migrate M_g amount of data, there will be $2(|S|-1)$ data transfer cycles, each containing $\lfloor \frac{|S|}{2} \rfloor$ parallel paths, and a total of $\big(|S|(|S|-1)\big)$ *mutually-exclusive* parallel sets of servers. To estimate the true parallelism during data migrations, a simple scheduling algorithm is presented in Algorithm 3.1 where the output M_g will be the true cost of serialised data migrations during a repartitioning cycle. The order of finding D_m can be derived from the order of sorting M (containing $|S|^2$ elements), which is $O(|S|^2 \log |S|)$.

In practice, it is the responsibility of the OS to perform such optimisation to maximise its parallelism for data transfer operations. However, to evaluate the performance of a repartitioning scheme in a simulated environment, it is sufficient to find an approximate value for pair-wise data migrations, which is very close to the true measure. A simple heuristic is to sum up the counts in every $\frac{\lfloor |S| \rfloor}{2}$th position of M to get the approximate parallelism during data migration operations. Thus, the estimated pair-wise

Algorithm 3.1 The algorithm to find parallelism for inter-server data migrations between *mutually-exclusive* server pairs

Input: Matrix $\mathcal{C}$, where c_{ij} is the count of data migrations between servers i and j
Output: M_g

```
1: procedure FINDPARALLELISM(C)
2:     M_{g_p} ← 0
3:     create ⟨K,V⟩, a sorted map, from C consisting of the mutually-exclusive server
           pairs and their corresponding counts in ascending order
4:     while ⟨K,V⟩ ≠ ϕ do
5:         select ⟨k_1,v_1⟩ and its mutually-exclusive pairs up to ⟨k_{n_e},v_{n_e}⟩ having
             minimum counts only
6:         select the maximum count v_max from the selected pairs
7:         M_g ← M_g + v_max
8:         remove all selected pairs from ⟨k_1,v_1⟩ up to ⟨k_{n_e},v_{n_e}⟩ from ⟨K,V⟩
9:     end while
10:    return M_g
11: end procedure
```

inter-server data data migrations can be defined as

$$\widetilde{M_g} = \sum_{i=1}^{\left\lceil \frac{|S|(|S|-1)}{\left\lfloor \frac{|S|}{2} \right\rfloor} \right\rceil} m_{\left\lfloor \frac{|S|}{2} \right\rfloor i}. \tag{3.15}$$

Finally, the normalised metric of inter-server data migrations can be redefined as

$$D_m = \frac{\widetilde{M_g}}{|\mathcal{D}_S|}. \tag{3.16}$$

Note that, the value of $\widetilde{M_g}$ will be naturally much smaller than $|\mathcal{D}_S|$; therefore, the impacts of this metric will be much lower in observation as well.

3.6 Conclusions

To define a workload-aware incremental repartitioning system, it is essential to define its core architecture, constructions, evaluation methods, and KPIs. In this chapter, a high-level overview of the proposed workload-aware incremental repartitioning framework is presented with detailed descriptions and functionalities of its key components. A novel distributed data lookup technique is introduced, which enables scalable

and consistent data lookup in a partitioned database, and allows elastic scale-out of the underlying OLTP database cluster. Furthermore, to demonstrate the usability of the proposed framework, a novel simulation platform is introduced, which contains an innovative transaction generation model for two mainstream OLTP workloads, representing business order-processing (TPC-C) and on-line social-networking (Twitter) application domains. The capability of the proposed transaction generation model in controlling transactional repetitions for generating a target proportion of unique transactions within an observation window is demonstrated through a sensitivity analysis. To evaluate the performance of the incremental repartitioning cycles, three unique KPIs—I_d, L_b, and D_m—are proposed, normalised, and explained with their physical interpretations. In next chapter, we present a family of graph *min-cut* based incremental repartitioning schemes and evaluate their repartitioning quality based on the proposed KPIs in this chapter.

Chapter 4

Incremental Repartitioning with Graph Theoretic Abstractions

In the previous chapter, the preliminaries of the proposed workload-aware incremental repartitioning framework are discussed, along with a novel transaction generation model for restricting unique transaction generations, and essential KPIs for evaluation purpose. In this chapter, a graph theoretic scheme for incremental repartitioning is presented which primarily focuses on reducing the adverse impacts of DTs in OLTP databases. The proposed scheme uses the graph based workload network clustering along with two cluster-to-partition mapping techniques that are proposed to ensure the balance between maintaining effective server-level data distribution and the amount of physical data migration.

4.1 Introduction

Among the incremental repartitioning strategies discussed in Chapter 2, the graph based scheme is the oldest, and by far the most applied technique in the literature. The primary reason for introducing a graph theoretic approach is to hide the underlying complexities of a transactional workload network by creating a higher level of abstractions. In this process, the transactions are naturally represented as a network of graph or hypergraph at first. Then, external graph *min-cut* libraries are used to produce balanced clusters by intersecting a minimum number of edges. Finally, the clusters are mapped back to the DSs by performing necessary data migrations. By imposing both *min-cut* and *balance* constraints, the graph *min-cut* based strategy indirectly reduces the overall impacts of DTs and produces similar sized clusters of tuples that should be placed together in a DS. Therefore, the minimisation of the load-balance KPI L_b comes as a by-product of the graph *min-cut* process, while there is no way to improvise D_m.

Hence, in a graph theoretic approach, there are too many limitations in controlling data distribution load imbalances with minimum data migrations. Furthermore, unless the graph or hypergraph networks are not of the same size, it is infeasible to compare and evaluate different graph theoretic repartitioning approaches.

In this chapter, a graph based approach for incremental repartitioning is used, where, during each cycle, transactional workloads are represented by either a graph or hypergraph network abstraction. This high-level network is then clustered using a balanced graph *min-cut* technique, and the clusters thus produced are mapped back to the logical database partitions instead of the physical DSs to ensure fine granularity. A novel transaction classification technique is introduced as well, which creates a sub-network, consisting of DTs and *moveable* NDTs containing at least a single tuple in a DT. Two special cluster-to-partition mapping techniques are proposed to ensure minimum physical data migrations by favouring the current partition of the majority tuples in a cluster—the many-to-one version thus minimises data migrations alone, and the one-to-one version reduces data migrations without affecting server-level load-balance. Experimental evaluations using *OLTPDBSim* shows the effectiveness of the proposed techniques by comparing different workload network representations, and cluster mapping techniques, thanks to our novel transaction generation model. Experimentations are conducted in a *proactive* (with hourly cycles) and a *reactive* manner, both aimed at maintaining a user-defined threshold.

The rest of the chapter is organised as follows: Section 4.2 presents an overview of the graph *min-cut* based incremental repartitioning scheme. The proactive transaction classification technique is illustrated in Section 4.3. Section 4.4 discusses different workload network representations and their formulations, followed by the k-way balanced clustering to the workload networks in Section 4.5. Three cluster-to-partition mapping strategies are illustrated with examples in Section 4.6. Section 4.7 has the detailed discussion of the experimental evaluations and findings. Finally, Section 4.8 summarises the chapter contributions and results.

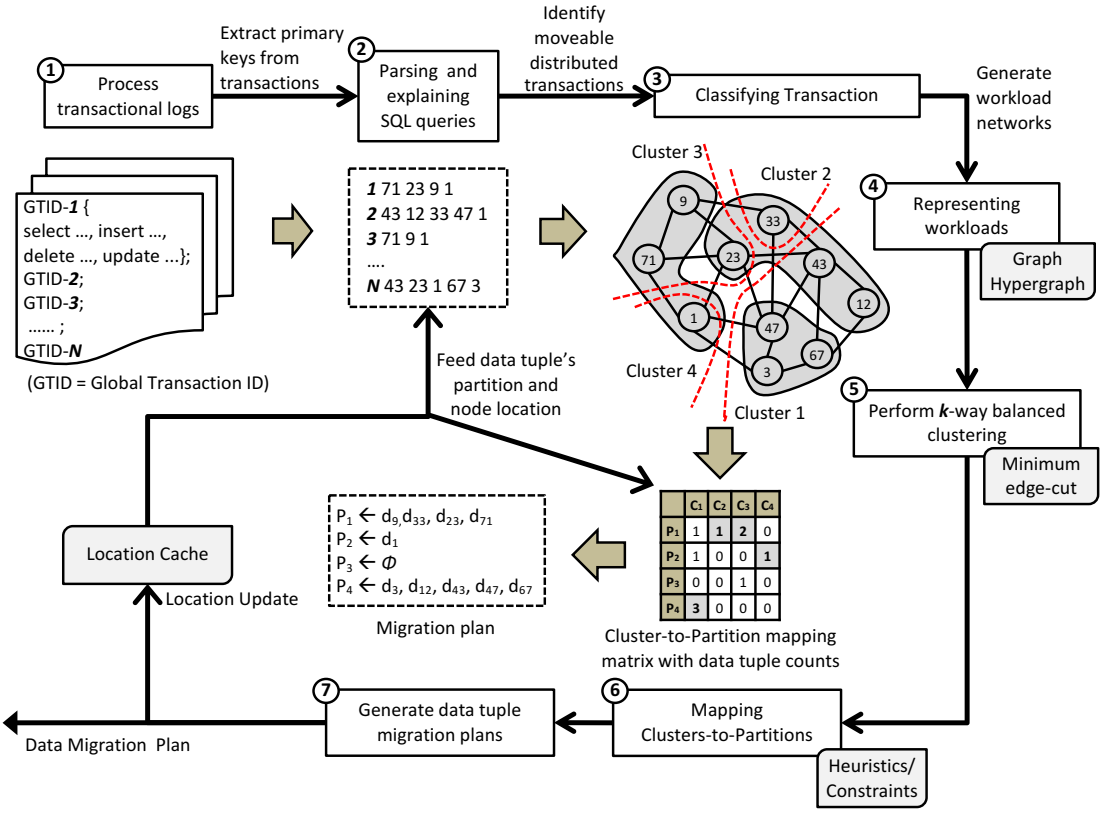


Figure 4.1: An overview of the graph *min-cut* based incremental repartitioning framework using numbered notations. Steps 1–7 represent the flow of workload analysis, representation, clustering, and repartitioning decision generation.

4.2 Overview

Figure 4.1 presents the graph *min-cut* based incremental repartitioning framework following steps 1–7. Recalling Figure 3.1, the input of the *Workload Analyser Node* is the transactional log streams and the output is a live data migration plan. The overall process has four primary activities as follows.

4.2.1 Transaction Pre-processing, Parsing, and Classification

Client applications submit database queries in step 1, which is then processed by a *Distributed Transaction Coordinator* that manages the execution of DTs within a *shared-nothing* DS cluster. Upon receiving the streams of transactional workloads, individual transactions are processed to extract the contained SQL statements in step 1. For each

SQL statement, the primary keys of individual tuples are extracted, and the corresponding partition ids are retrieved from the embedded workload specific location catalogue in step 2. In the classification process in step 3, DT and NDTs are identified along with their frequency counts in $\mathcal{W}$, and their associated costs of spanning multiple servers.

4.2.2 Workload Network Representations and k-way Clustering

In step 4, workload networks are generated from the extracted transactional logs gathered in the previous step, using graphs or hypergraphs. Tuple-level compression can further reduce the size of the workload network. Since simple graphs can not fully represent transactions with more than two tuples using a pair-wise relationship, it is not possible to minimise the impacts of DTs in the workload directly. However, graph representations are much simpler to produce, and they have been adopted in a wide range of applications that also help us understand its importance in creating workload networks. On the other hand, hypergraphs can exploit exact transactional relationships, thus the number of hyperedge-cuts exactly matches the number of DTs. Yet, popular hypergraph clustering libraries are computationally slower than the graph clustering libraries, and they produce less effective results [101]. In reality, with the increase in size and complexity, both of these representations are computationally intensive to manipulate. Furthermore, while compression techniques may address this problem, dramatic degradation in clustering quality and overall load-balance occur with a high compression ratio [125]. Finally, the workload networks are clustered using *k min-cut* graph and hypergraph clustering libraries in step 5.

4.2.3 Cluster-to-Partition Mapping

In step 6, a mapping matrix is created with the counts for tuples that are placed in the newly created cluster and that originated from the same partition as the matrix element. The clusters produced from the *min-cut* clustering are then mapped to the existing set of logical partitions by following three distinct strategies. At first, a uniform random tuple distribution is used for mapping clusters to database partitions which naturally balances the distribution of tuples over the partitions. However, there are no proactive considerations in this random strategy for minimising data migrations.

The second strategy employs a straightforward but optimal approach. It maps a cluster to a respective partition which originally contains the maximum number of tuples from that cluster, hence minimum physical data migrations take place. In many cases, this simple strategy turns out to be many-to-one cluster-to-partition mapping and it diverges from uniform tuple distribution. Again, incremental repartitioning can create server hot-spots as similar transactions from the latest $\mathcal{W}$s will always drive more new tuples to migrate into a hot server. As a consequence, overall load-balance decreases over time, which is also observed in our experimental results.

A way to recover from this situation is by ensuring that cluster-to-partition mapping remains one-to-one, which is used as the third strategy. This simple, yet effective, scheme restores the original uniform random tuple distribution with the constraint of minimising data migrations. Finally, in step 7, based on different mapping strategies and applied heuristics, a data migration plan is generated, and then forwarded to the *Data Tuple Migrator* component located in the *Workload Analyser Node*.

4.2.4 Distributed Location Update and Routing

The analyser node caches the most frequently accessed tuples in a workload specific location catalogue, and updates the associated locations at each repartitioning cycle. Until a tuple fully migrates to a new partition, its existing partition serves all the query requests. As discussed in Section 3.2.4, and depicted in Figure 3.2, the *lookup table* in each data partition keeps track of the *roaming* tuples and their corresponding *foreign* partitions. A maximum of two lookups are, therefore, required to find a tuple without client-side caching. With proper caching enabled at the *Workload Analyser Node*, this lookup cost can be even amortised to one for most of the cases with high cache hits.

4.3 Proactive Transaction Classification

Typical graph based repartitioning schemes only use DTs for *min-cut* clustering purposes. However, there always exist NDTs that contain one or more tuples which also reside in a DT. During the repartitioning process, these NDTs may turn into DT if they are

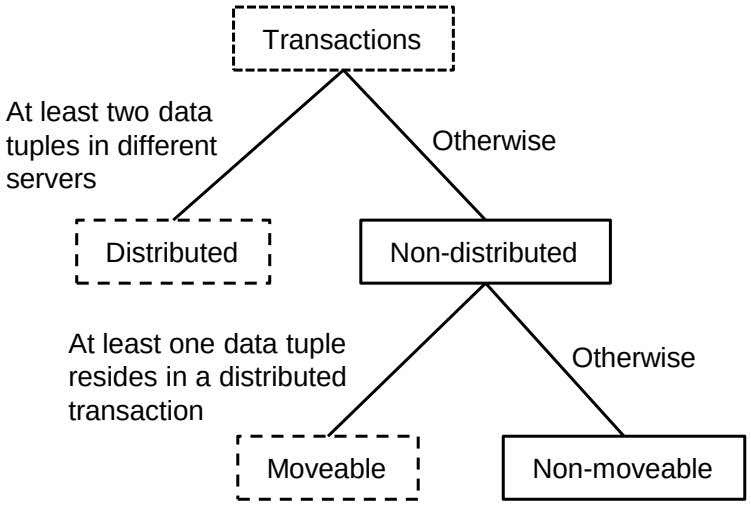


Figure 4.2: The proactive transaction classification tree identifying DTs, *moveable* NDTs, and NDTs

Table 4.1: Proactive transaction classification of the sample workload

Id	Tuples	Class
τ_1	{1, 4, 5, 6, 7, 8, 10}	DT
τ_2	{1, 4, 6, 9, 11}	DT
τ_3	{9, 15, 17}	MNDT
τ_4	{9, 17}	MNDT
τ_5	{5, 7, 18}	DT
τ_6	{15, 17}	NMNDT
τ_7	{2, 14, 16}	NMNDT

not initially included in the workload network construction. This simple intuitive observation is used to classify the workload transactions into three different categories—DTs, *moveable* NDTs (MNDTs), and NDTs i.e., *non-moveable* (NMNDT)—as shown in Figure 4.2. For illustration purposes, the sample shared-nothing distributed database cluster from Table 2.4 is considered. Transactions listed in Table 2.5 are reused here to illustrate the different transaction classes described above. Here, transactions τ_1, τ_2, and τ_5 from the sample workload are identified as DTs. Transactions τ_3 and τ_4 are labelled as MNDTs as both contained data tuple 9, which also resides in DT τ_2. Finally, τ_6 and τ_7 are discarded as purely NDT transactions as none of their data tuples resides in any DT. The classified transactions are re-listed in Table 4.1.

The philosophy behind the above idea is quite intuitive. For a large graph, there

may be many k-way *min-cut* clustering solutions, all inducing the minimum number of edge-cuts. When a transactional workload graph is constructed using only the DTs, one of the many possible k-way clustering solutions is picked, at random or based on some other criteria, by the graph-cut algorithm in use. Consequently, the repartitioning scheme has no means of protecting a NDT from becoming a DT after the repartitioning, even when opportunities exist, unless all the clustering solutions are checked explicitly, which is infeasible. If MNDTs are included in the graph with additional edges, a *min-cut* solution will not cut one of these edges unless absolutely necessary, i.e., only when all other possible k-way clustering options require cutting more edges. Effectively, the repartitioning scheme is then able to protect implicitly NDTs from becoming DTs after repartitioning when an alternative clustering opportunity exists.

Note that, the addition of an extra set of MNDTs increases the workload sub-network size for clustering purposes. Therefore, there is a clear trade-off between the increase in size of the workload networks and the benefits achieved. The smaller the sub-network, the less costly it is, in terms of computations, processing, and IO. Alternatively, if all the current and future DTs are given to the *min-cut* clustering process, I_d might be reduced further during a particular repartitioning cycle. By aggressively classifying the *moveable* NDTs, the quality of the overall repartitioning process increases as the impacts of DTs decreases, compared to a static repartitioning strategy, as validated later in the experimental results.

4.4 Workload Network Representations

Following the transaction classification, a workload network is created from the selective set of transactions as either a graph, a hypergraph, or one of their compressed forms as discussed in Section 2.5. The workload networks are modelled under four distinct representations. Firstly, *Graph Representation (GR)* produces a fine-grained workload network, although it is unable to capture completely the actual transactional relationship between different tuples. Despite the shortcomings, a graph clustering process can still produce high quality clusters for the repartitioning schemes to minimise I_d. However, this quality can only be maintained as long as the graph size increases with

the workload volume and at least the necessary level of sampling is performed [101]. Secondly, *Hypergraph Representation (HR)* can represent the workload transactions in an accurate manner. In empirical studies, it is found that a k-way *min-cut* balanced hypergraph clustering process produces more consistent repartitioning outcomes than graphs [116]. Finally, *Compressed Graph Representation (CGR)* and *Compressed Hypergraph Representation (CHR)* produces coarse-grained workload networks depending on the compress level. With a lower level of compression, less coarse networks are generated and k-way clustering performs better. However, as shown in [125], as the level of compression increases the quality of the clustering process degrades dramatically. We turn now to defining the individual network representation formally in the following subsections.

4.4.1 Graph Representation

Let $V = \{v_1, ..., v_{|V|}\} \subset \mathcal{D}_S$ be the set of all data tuples collectively used in all DTs and MNDTs in the transactional workload in an observation window $\mathcal{W}$. By using these data tuples as vertices, a graph representation *GR* $\mathcal{G} = (V, E_g)$ may be constructed where an edge between two vertices exists only if both data tuples are used by a DT or a MNDT, i.e., $(v_x, v_y) \in E_g$ only when $\exists\tau$ among the DTs and MNDTs such that $\{v_x, v_y\} \subseteq \tau$, for all $1 \leq x \leq y \leq |V|$. A transaction τ using l data tuples is represented in this graph by the 'complete' subgraph of the respective vertices having all possible $\binom{l}{2} = \frac{l(l-1)}{2}$ pair-wise edges. Note the many-to-many relationship between edges and transactions. An edge of the graph may be used to represent multiple transactions and vice versa. In fact, two transactions τ_a and τ_b share $\binom{|\tau_a \cap \tau_b|}{2}$ edges altogether. Such a convoluted relationship may render the abstraction at least confusing and, therefore, the graph representation is considered not ideal for our purposes as alluded to earlier. A weighted graph is used by the k-way balanced *min-cut* algorithm. Weight w_{v_x,v_y} of each edge $(v_x, v_y) \in E_g$ is assigned as the sum of the frequencies of all related transactions, i.e., $w_{v_x,v_y} = \sum_{\forall\tau|\{v_x,v_y\}\subseteq\tau} f(\tau)$. Weight of each vertex represents the data size (in volume) of the corresponding tuple.

4.4.2 Hypergraph Representation

A hypergraph representation *HR* $\mathcal{H} = (\mathcal{V}, E_h)$ of the same transactional workload uses the same set V of vertices used for the equivalent graph representation. The pair-wise edge set, however, is replaced with a set of hyperedges, each uniquely representing a DT or a MNDT with a one-to-one relationship. Just like a transaction, a hyperedge is a subset of V, which is drawn with a bounded region covering all, and only, the vertices used in the corresponding transaction. Again, a weighted hypergraph is used by the k-way balanced *min-cut* algorithm where weight w_τ of each hyperedge $\tau \in E_h$ is assigned as the frequency of representing transaction τ, i.e., $w_\tau = f(\tau)$. The weight of each vertex remains the same, i.e., the data size (in volume) of the corresponding tuple.

4.4.3 Compressed Representation

Both the graph $\mathcal{G} = (V, E_g)$ and hypergraph $\mathcal{H} = (V, E_h)$ representations of the transaction workload can be compressed by collapsing the vertices onto a set of virtual vertices $V' = \{v'_1, ..., v'_{|V'|}\}$ using a simple hash function on the primary keys [125]. Effectively, virtual vertices are disjoint subsets of V that cover it completely, i.e., $v'_i \subset V$, for all $1 \leq i \leq |V'|$, such that $v'_i \cap v'_j = \phi$, for all $1 \leq i < j \leq |V|$, and $v'_1 \cup ... \cup v'_{|V|} = V$. A compressed graph representation *CGR* $\mathcal{G}_c = (V', E'_g)$, equivalent to graph $\mathcal{G} = (V, E_g)$, has a compressed edge $(v'_i, v'_j) \in E'_g$ only if $\exists v_x \in v'_i$ and $\exists v_y \in v'_j$ such that $(v_x, v_y) \in E_g$. Similarly, a compressed hypergraph representation *CHR* $\mathcal{H}_c = (V', E'_h)$ equivalent to hypergraph $\mathcal{H} = (V, E_h)$ has a compressed hyperedge $e'_h \in E'_h$ only if $\exists e_h \subset e'_h$ such that $e_h \in E_h$. Weights of the virtual vertices and edges (hyperedges) of a compressed graph (compressed hypergraph) are set by adding all associated weights in the corresponding graph (hypergraph). Note that different graphs (hypergraphs) may be collapsed onto the same compressed graph (compressed hypergraph), leading to a many-to-one equivalence relationship. Also note that the ratio $C_l = \frac{|V|}{|V'|}$ denotes the compression level, ranging from 1 (no compression), when $|V'| = |V|$, to $|V|$ (full compression), when $|V'| = 1$.

4.5 *k*-way Balanced Clustering of Workload Network

Given $\mathcal{G}$ and a maximum allowed imbalance fraction ε, the k-way clustering of $\mathcal{G}$ can be defined as $\zeta_{\mathcal{G}} = \{V_1, ..., V_k\}$, which thus minimises *edge-cuts* within the load *balance* factor tolerance range. Similarly, the k-way constrained and balanced clustering of $\mathcal{H}$ is $\zeta_{\mathcal{H}} = \{V_1, ..., V_k\}$, such that a minimum number of hyperedges are cut within the imbalance fraction ε. Analogously, the k-way balanced clusterings of $\mathcal{G}_c$ and $\mathcal{H}_c$ are $\zeta_{\mathcal{G}_c} = \{V'_1, ..., V'_k\}$ and $\zeta_{\mathcal{H}_c} = \{V'_1, ..., V'_k\}$, respectively, with an imbalance fraction ε and aiming at minimum compressed hyperedge-cuts. Note that, k is denoted as the total number of logical partitions instead of the number of physical servers. From our empirical studies, it is observed that executing the k-way clustering with k as the number of partitions provides finer granularity in balancing the data distribution over the set of physical servers.

When a set V of a large number of data tuples is divided into k disjoint subsets (clusters) $V_1, ..., V_k$, the load-balancing factor of the clustering may be calculated as

$$\beta_k = \frac{k \max_{1 \leq i \leq k} w(V_i)}{\sum_{i=1}^{k} w(V_i)} \tag{4.1}$$

where $w(V_i)$ is the collective data size (in volume) of all tuples in cluster V_1, for all $1 \leq i \leq k$. Note that $\beta_k \geq 1$ and the perfect load-balancing factor of 1 is achieved when $w(V_1) = ... = w(V_k)$. Often, to provide some flexibility in a system, the load-balancing factor is constrained to a range $[1, 1+\varepsilon]$, where ε is a small real number, denoting the maximum fraction of imbalance tolerated by the system. Figures 4.4 and 4.5 present the k-way balanced clustering of different workload network representations of graph and compressed graph, and hypergraph and compressed hypergraph, respectively.

Figure 4.3 shows the resulting 2-way *min-cut* clustering of the sample transactions classified by the proposed proactive transaction classification tree which includes the MNDTs τ_3 and τ_4. One can easily observe and compare the differences between the the clustering performed without including τ_3 and τ_4 in Figure 2.9. For instance, we have 3 DTs after the *HGR* clustering without including the MNDTs whereas in Figure 4.3 we can manage to convert one DT to an NDT as well as save the MNDT from becoming an DT. This signifies the effectiveness of our proposed transaction classification technique.

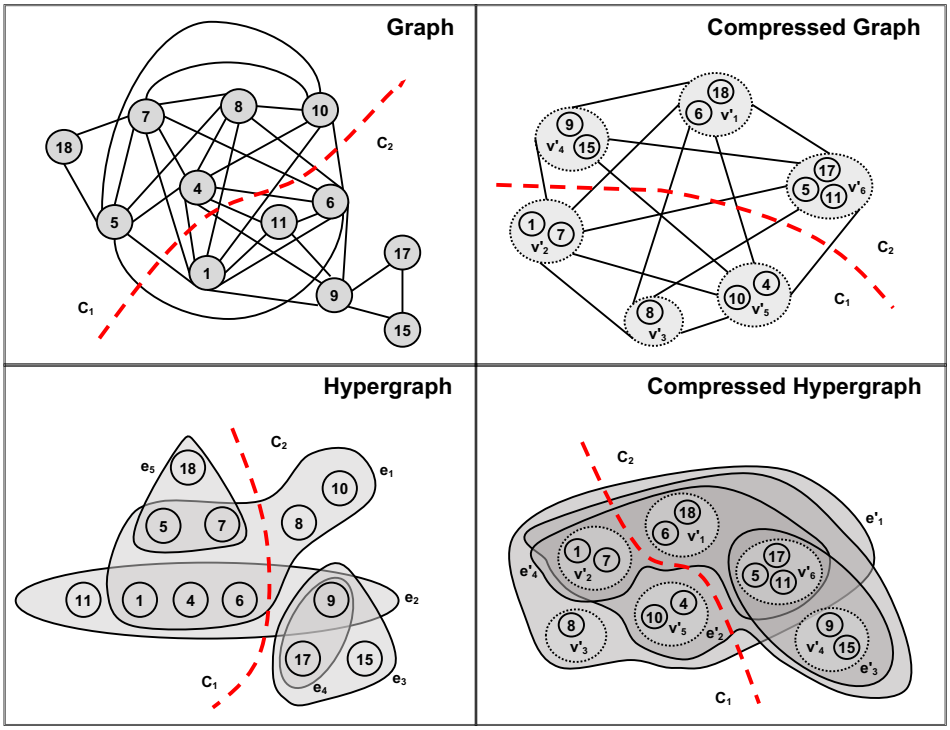


Figure 4.3: The 2-way *min-cut* clustering of the sample transactional workload network represented as graph, hypergraph, compressed graph, and compressed hypergraph using the proactive transaction classification tree

4.6 Cluster-to-Partition Mapping Strategies

Following the *k*-way balanced *min-cut* process, next, the clusters need to mapped back to the database. As mentioned earlier, for mapping purposes the logical partitions are used as the targets instead of the physical servers, firstly, to ensure finer granularity, as the number of partitions are always much higher than the number of DSs in the cluster, and secondly, for supporting our proposed distributed data lookup technique which uses partition level lookup tables for both *range-* and *consistent-hash partitioned* databases. In the following, illustrative examples are provided using the simple database setup shown in Table 2.4 with 20 data tuples distributed over four logical partitions and two physical DSs. A sample workload batch with seven transactions and corresponding data tuples are also shown in Table 4.1. Figures 4.4 and 4.5 present a detail illustration of how the proposed cluster-to-partition mapping strategies work with different workload representations.

In these examples, workload networks are represented as *GR*, *CGR*, *HR*, and *CHR*

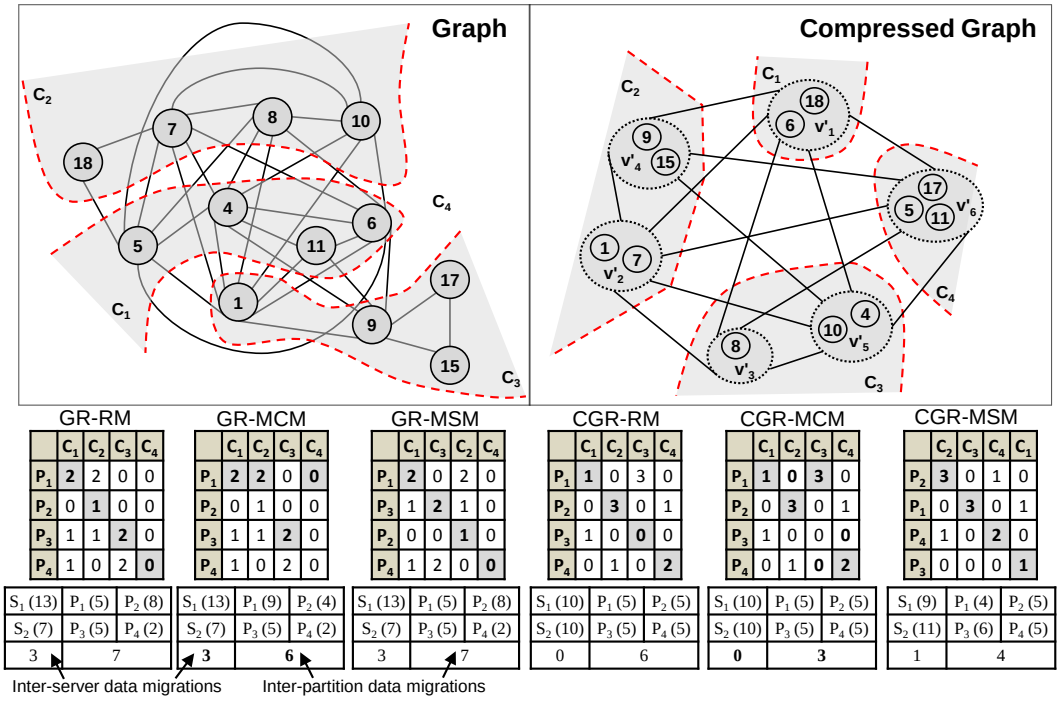


Figure 4.4: The *k*-way *min-cut* clustering of graph and compressed graph representations of the workload networks followed by 3 cluster-to-partition mapping strategies

(with $C_l = 2$) for the transactions listed in Table 4.1. Three distinctive cluster-to-partition mapping strategies (in matrix format) are also shown below their respective workload network representations. The rows and columns of the matrices represent partition and cluster ids, respectively. An individual matrix element represents tuple counts from a particular partition which is placed by the clustering libraries under a specific cluster id. The shadowed locations in the mapping matrices with the counts in *bold font* represent the resulting decision blocks with respect to the particular cluster and partition id. Individual tables below the matrices represent the state of the physical and logical layouts of the sample database. The last row of these tables reveals the counts of inter- and intra-server data migrations for each of these nine representative database layouts. The *bold font* numbers in the layout tables at the bottom denote most balanced data distributions and minimum inter-server and inter-partition data migrations. The details of the individual mapping techniques are discussed as below.

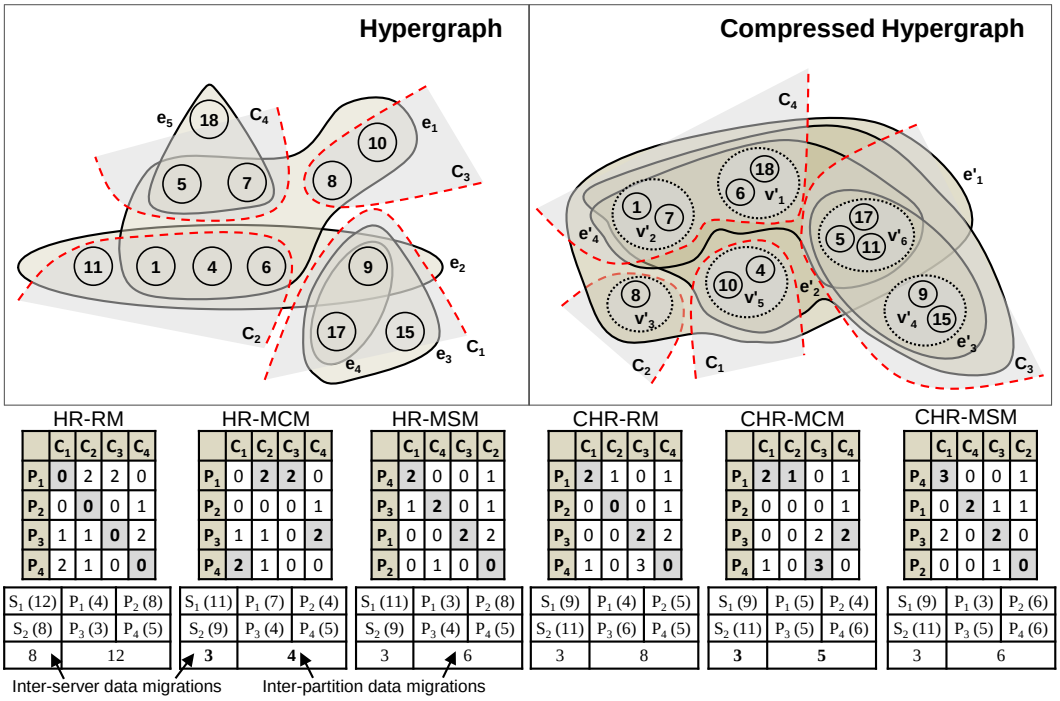


Figure 4.5: The *k*-way *min-cut* clustering of hypergraph and compressed hypergraph representations of the workload networks followed by three cluster-to-partition mapping strategies

4.6.1 Random Mapping (RM)

Naturally, the best way to achieve load-balance in any granularity is to assign the clusters randomly. Graph clustering libraries like *METIS* [87, 89] and *hMETIS* [85, 90] randomly generate the cluster ids, and do not have any knowledge as to how the data tuples are originally distributed within the servers or partitions. As a straightforward approach, the cluster ids can be simply mapped one-to-one to the corresponding partition id as they are generated. Although this random assignment balances the workload tuples across the partitions, it does not necessarily guarantee minimum inter-server data migrations. As shown in Figures 4.4 and 4.5, the mapping matrices labelled with *RM* and database layouts with *GR-RM*, *CGR-RM*, *HR-RM*, and *CHR-RM* are the representatives of this class, respectively.

4.6.2 Maximum Column Mapping (MCM)

In this mapping, the aim is to minimise the inter-server physical data migration within a repartitioning cycle. In the cluster-to-partition mapping matrix, the maximum

tuple count of an individual column is discovered, and the entire cluster column is mapped to the representative partition id of that maximum count. If multiple maximum counts are found, then the one directing the partition containing the lowest number of data tuples is chosen. Thus, multiple clusters can be assigned to a single partition. As the maximum numbers of tuples are originated from this designated partition, they do not tend to leave from their *home partitions* which reduces the overall inter-server physical data migrations. For OLTP workloads with skewed tuple distributions and dynamic data popularity, the impacts of DTs can rapidly decrease from this greedy heuristic as tuples from multiple clusters may map into a single partition in the same physical DS. However, this may lead to data distribution imbalance across the partitions as well as the servers, despite our selection preference for the partition id in the mapping matrix. Mapping matrices labelled as *MCM* with corresponding database layouts of *GR-MCM* and *CGR-RM* in Figure 4.4, and *HR-MCM* and *CHR-MCM* in Figure 4.5, represent this mapping, respectively.

4.6.3 Maximum Submatrix Mapping (MSM)

In this final mapping technique, the natural advantages of the previous strategies are combined in order to minimise load-imbalance and data migrations simultaneously. At first, the largest tuple counts within the entire mapping matrix are found and placed at the diagonally top left position by performing successive row-column rearrangements. The next phase begins by omitting the elements in the first row and column, then recursively searching the remaining *submatrices* for the element with the maximum tuple counts. Finally, all the diagonal positions of the matrix are filled up with elements having maximum tuple counts. Mapping the respective clusters one-to-one to the corresponding partitions in such this way results in both minimum data migrations and distribution load-balance. Note that, multiple maximum tuple counts can be found in different matrix positions, and the destination partition id is chosen randomly from any of these places. The *MSM* strategy works similarly to the *MCM* strategy as it prioritises the maximum tuple counts within the submatrices, and maps the clusters one-to-one to the partitions like the *RM* strategy, thus preventing potential load-imbalance across both the logical partitions and physical servers. In Figures 4.4 and 4.5, mapping

matrices labelled as *MSM*, and representative database layouts *GR-MSM*, *CGR-MSM*, *HR-MSM*, and *CHR-MSM*, represent this mapping strategy, respectively.

4.7 Experimental Evaluation

Both *range* and *consistent-hash* based initial data partitioning are used to evaluate four different scenarios to evaluate the proposed graph theoretic incremental repartitioning schemes. The transaction generation model integrated with *OLTPDBSim* is configured to use a restricted repetition window for generating a target proportion of unique transactions to keep the generated workload network sizes the same across all other settings. Table 4.2 lists the related simulation parameters. Both *range*- and *consistent-hash partitioned* OLTP databases are simulated in *OLTPDBSim* following two strategies: 1) no repartitioning or static (i.e., a single) repartitioning, and 2) incremental repartitioning—hourly or based on a predefined repartitioning triggering threshold R_t. The *No Repartitioning (NR)* and *Static Repartitioning (SR)* (which will behave very similarly to *Schism* [101]) schemes are considered as the baseline for comparing *Hourly Repartitioning (HR)*, and *Threshold-based Repartitioning (TR)* based incremental repartitioning strategies. In *SR*, the database is repartitioned only once after the database is warmed up, and then remains static for the rest of its lifetime. As a proactive approach, the *HR* process repartitions the database at the end of each observation window, regardless of whether the systemwide I_d is below R_t or not. This eventually reveals the highest ability of any data migration strategy to reduce the impacts of DTs in an incremental manner. Finally, *TR* only works when the per transaction per unit time I_d increases over the triggering threshold. Following a repartitioning cycle the *TR* sets back a preset cooling-off period of one hour and does not trigger another repartitioning phase within this time. Furthermore, *METIS* [87, 89] and *hMETIS* [85, 90] k-way *min-cut* clustering libraries are used with their default settings. These comprehensive experimentations help in understanding the applicability of incremental repartitioning within a wide variety of settings. The effectiveness of the proposed techniques is evaluated with respect to I_d, L_b, and D_m (as detailed in Chapter 3) for consecutive repartitioning cycles.

Table 4.2: The key simulation configuration parameters defined in *OLTPDBSim*

Workload profile	TPC-C (at scale of 0.1 with 10 warehouses) Twitter (at scale of 500 users)
Database tables	9 for TPC-C 4 for Twitter
Number of physical servers, S	4
Partition assignment to server	Round-robin
Initial data partitioning	*Range* with 36 Partitions *Consistent-hash* with 16 Partitions
Workload modelling	*GR*, *CGR*, *HR*, *CHR*
Cluster-to-Partition mappings	RM, MCM, MSM
$\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p \rangle$	$\langle 1, 0.25, 3600, 0.15 \rangle$
Repartitioning triggering threshold, R_t	$2/S$
Repartitioning cooling off period	1 h
Transactional repetition pool	Restricted
Exponential averaging weight, α	0.6
Compression level, C_l	6
Repartitioning schemes	*NRP*, *SRP*, *HRP*, *TRP*
Database warm-up period	3 h
Database operational period	24 h

4.7.1 Analysis of Experimental Results for *TPC-C* and *Twitter* Workloads

The distributed OLTP databases are simulated with two different initial data partitioning schemes: 1) *range*, and 2) *consistent-hash*. The following definitions of *range* and *consistent-hash* schemes are adopted for initial data partitioning in the simulation. *Range* scheme partitions each database table into a number of logical partitions with fixed volume size based on the total number of physical servers allocated. When data volume reaches its bound a partition split takes place to create two new equal sized logical partitions, and later the partition management module can redistribute the partitions for load-balance purposes. *Consistent-hash*, on the other hand, starts with a predefined number of logical partitions with a very large data volume limit. For example, a *consistent-hash* ring made with 'SHA-1' has a bit space 2^{160}, and with 16 fixed logical partitions each holding up to $2^{160}/16$ data rows of a partitioned database. In terms of scalability management, a new DS can be easily added by redistributing only

a fixed amount of data tuples within the cluster while a *consistent-hash* scheme is used. The data lookup process is also distributed and highly scalable as the *consistent-hash* function never changes. In contrast, a centralised lookup server is required for a *range* scheme and a large volume of data migrations is required while adding a new server in the system. By adopting the proposed distributed data lookup method based on the concept of *roaming* (as proposed in Section 3.2.4), both of these data partitioning schemes are included in the simulation model.

To distinguish clearly the differences between different incremental repartitioning schemes and data migration strategies, workload variations are not induced during the lifetime of the simulated databases. Tuple-level data replications are also limited to only 1 (i.e., no replication) so that the actual performance of graph or hypergraph *min-cut* based clustering can be revealed. Four different repartitioning scenarios are compared in the experiments conducted. Without adopting any repartitioning scheme, the underlying database cluster maintains a consistent I_d over successive time periods. Applying any repartitioning scheme during this steady-state period triggers data redistributions to minimise the value of I_d, and the goal in these experimental evaluations is to find out how a static, proactive, and reactive repartitioning approach reacts into such situation. Furthermore, the results explain the overall behaviour of using a graph theoretic repartitioning scheme which generally abstracts the workload complexities by using higher level of abstractions. The performance of 12 different incremental repartitioning schemes are examined under four categories for both *range-* and *consistent-hash partitioned* databases. The user-defined repartitioning triggering threshold R_t is set to $2/|S|$ as per (3.9): that means R_t is set such that on average each transaction only spans half of the DSs, thus maintaining an overall balance for physical resource usages. Below the repartitioning KPIs of the proposed schemes and their variations are compared. The progressions of I_d values are observed in line plots (by applying appropriate line smoothing), while I_d, L_b, and D_m are presented in the box plots as well.

4.7.1.1 Range-Partitioned Database

Figure 4.6 demonstrates the impacts of DTs for a *range-partitioned* database under *TPC-C* workload. As observed, the value of I_d remains around 0.7 for the entire lifetime

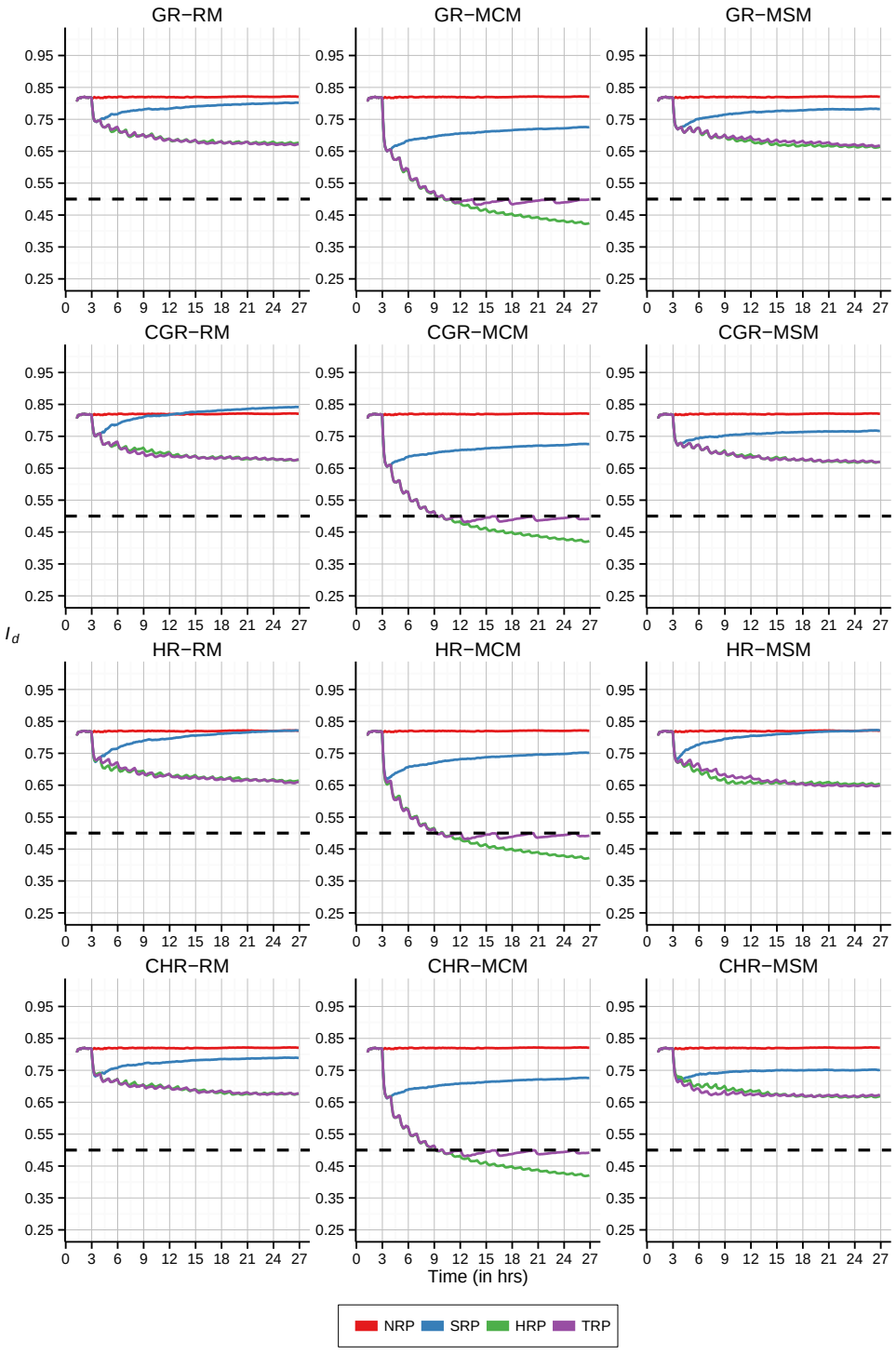


Figure 4.6: Comparison of different incremental repartitioning schemes in a *range-partitioned* database cluster for observing the variations of I_d under *TPC-C* workload

of the database if no repartitioning cycle is carried out. This is the worst-case scenario and the baseline in accordance with the usual transaction processing operations of the underlying database. A single repartitioning cycle (i.e., *SRP*) can hold down the value of I_d for a certain amount of time, after which it gradually increases again over time. For *GR-MSM*, the increase in I_d beyond 0.7 indicates that the *SRP* strategy is worse than having no repartitioning cycles at all, at least for some cases. Both *HRP* and *TRP* strategies for *RM* and *MSM* data migrations reduce the value of I_d within the range of 0.65–0.60 but not below that. Hence, it is the maximum ability of these repartitioning schemes to hold down the increase of I_d by redistributing data tuples within a database for *TPC-C* workload. On the other hand, *MCM* based strategies are the only incremental repartitioning schemes that are able to hold down I_d below the R_t margin. As *MCM* implements a many-to-one cluster-to-partition data migration strategy which is simultaneously applied over time, *HRP* tends to decrease I_d up to its lower bound. *TRP* for *MCM* strategies, on the other hand, works as anticipated and performs incremental repartitioning whenever I_d suppresses R_t. For all the *MCM* strategies under this scheme have nine incremental repartitioning cycles over the period of 24 h under *TPC-C* workload.

In the case of *Twitter* workload as shown in Figure 4.7, the value of I_d remains above 0.95 with *NRP*. As the workload presents complex network relationships, the performance of incremental repartitioning depends greatly on how it is actually represented to the graph *min-cut* libraries. Furthermore, depending on how effective the initial data redistribution is, the overall performance of a repartitioning scheme increases or decreases over repetitive cycles over time. Considering the *RM* data migration strategy, the *GR* and *CGR* based *SRP*, *HRP*, and *TRP* schemes consistently fluctuate between the I_d range of 0.55–0.65. On the other hand, the *HR* and *CHR* based schemes fall below 0.35 from R_t, which indicates that most of the workload data tuples are now concentrated in a single DS within the cluster, a situation which could lead to data distribution imbalance. Similarly, *MCM* based strategies for all of the repartitioning schemes steadily fall below R_t. In the case of the *MSM* based strategy, however, it shows similar characteristics to *RM*. While the *HGR* and *CHR* based repartitioning schemes consistently fall below R_t, the *GR* and *CGR* based *HRP* and *TRP* techniques gradually decrease the overall I_d of the system. *CGR-MSM*, in particular, performs well for both *HRP* and *TRP*

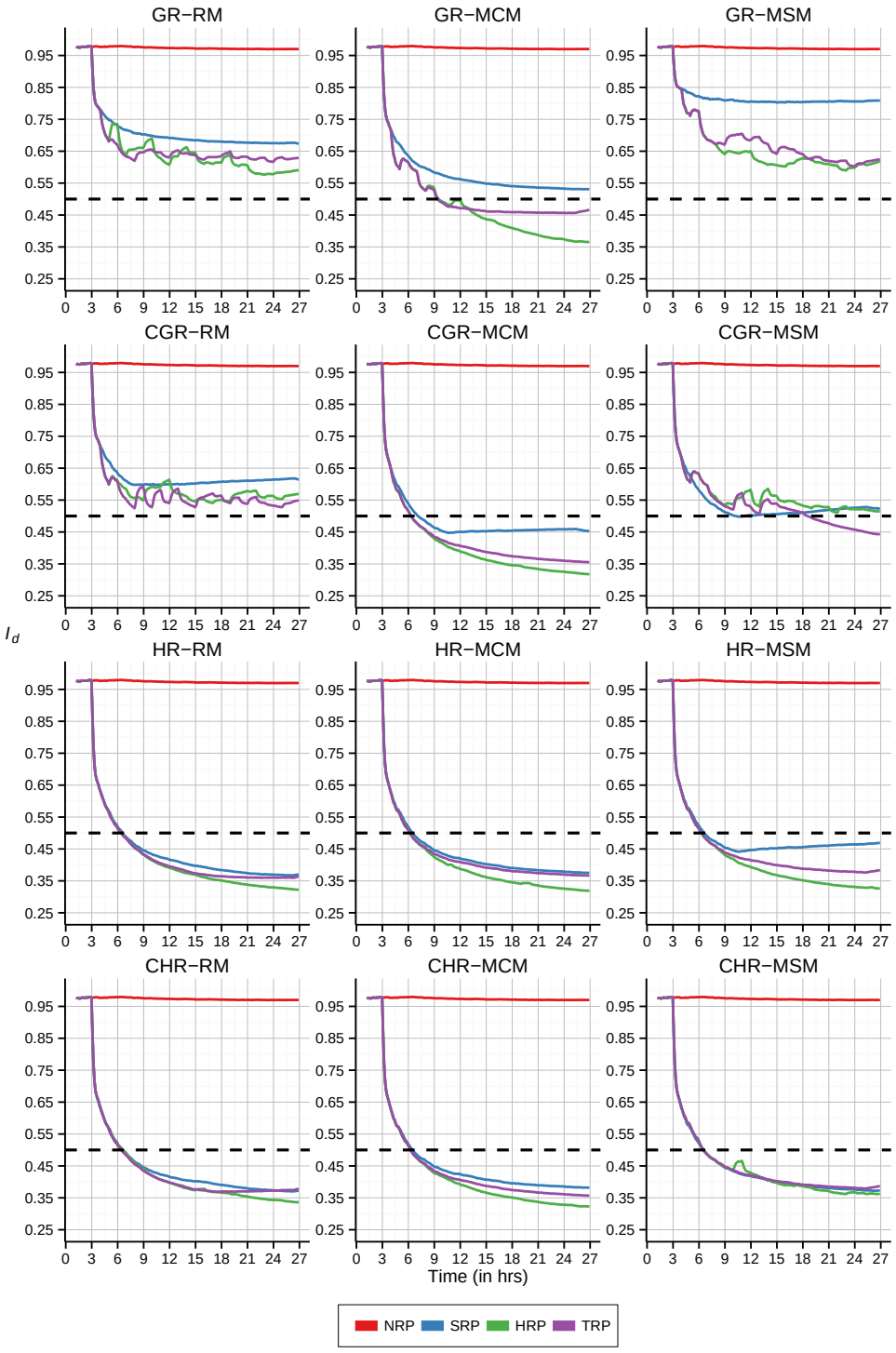


Figure 4.7: Comparison of different incremental repartitioning schemes in a *range-partitioned* database cluster for observing the variations of I_d under *Twitter* workload

to maintain a consistent I_d value near the R_t margin.

Figure 4.8 uses box-plots to present the simulation statistics for I_d, L_b, and D_m for *TPC-C* workload. From the results, it is clear that both *RM* and *MSM* based strategies maintain the load-balance for all workload representations for both *HRP* and *TRP*. In the case of *MCM*, severe load imbalance occurs for *HRP* due to the tendency to migrate more and more data tuples into the same DS over the incremental repartitioning cycles. Initial *range* partitioning also produces uneven logical data partitions, which also play a role as the decision of data migration is made at partition level. *TRP*, on the other hand, performs better; thus the mean values for all workload representations are close to *SRP*. In terms of inter-server data migrations, both *RM* and *MSM* strategies perform similarly, although the latter performs slightly fewer data migrations than the former. For all the cases of *MCM*, data migrations are very low, although the *TRP* scheme has visibly higher level fluctuations compared to the *HRP*. Overall, the proposed *MCM* strategy in combination with *TRP* works well for the *range-partitioned* OLTP database cluster under *TPC-C* workload.

In all cases of *Twitter* workload, as shown in Figure 4.9, the load-balance performance is strictly maintained for the *RM* and *MSM* based repartitioning schemes. The consistent fall of I_d below the R_t margin, as shown in Figure 4.7, and as anticipated, does not significantly contribute to the load imbalance for the *HR* and *CHR* based *HRP* and *TRP* schemes. Further investigation reveals that this is due to a few of the highly popular data tuples within the workload which significantly contribute to I_d, and which are later concentrated in a single DS during incremental repartitioning cycles. The *MCM* based *HRP* and *TRP* schemes for *HR* and *CHR* are primarily affected by load imbalance, while *GR* and *CGR* perform steadily in most of their cases. In case the of D_m, the *RM* and *MSM* based techniques perform more data migrations to maintain their load-balance compared to *MCM*. Among the *proactive HRP* schemes data migrations are less than, in the *reactive TRP* approaches. Furthermore, the *GR* and *CGR* based repartitioning schemes need to carry out fewer data migration operations comparing to *HR* and *CHR* based schemes for *Twitter* workload, in particular.

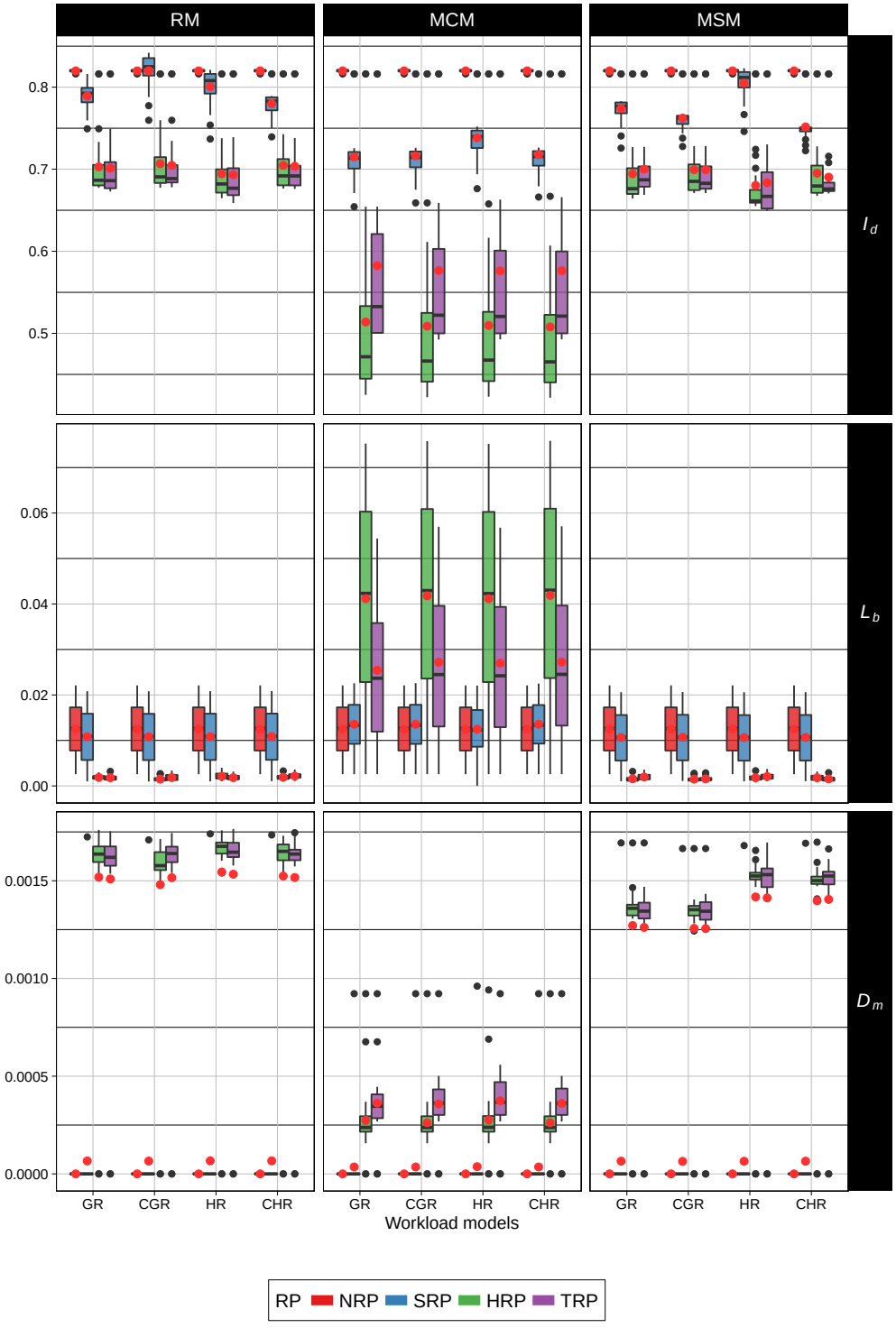


Figure 4.8: Comparison of different incremental repartitioning schemes in a *range-partitioned* database cluster for observing I_d, L_b, and D_m under *TPC-C* workload

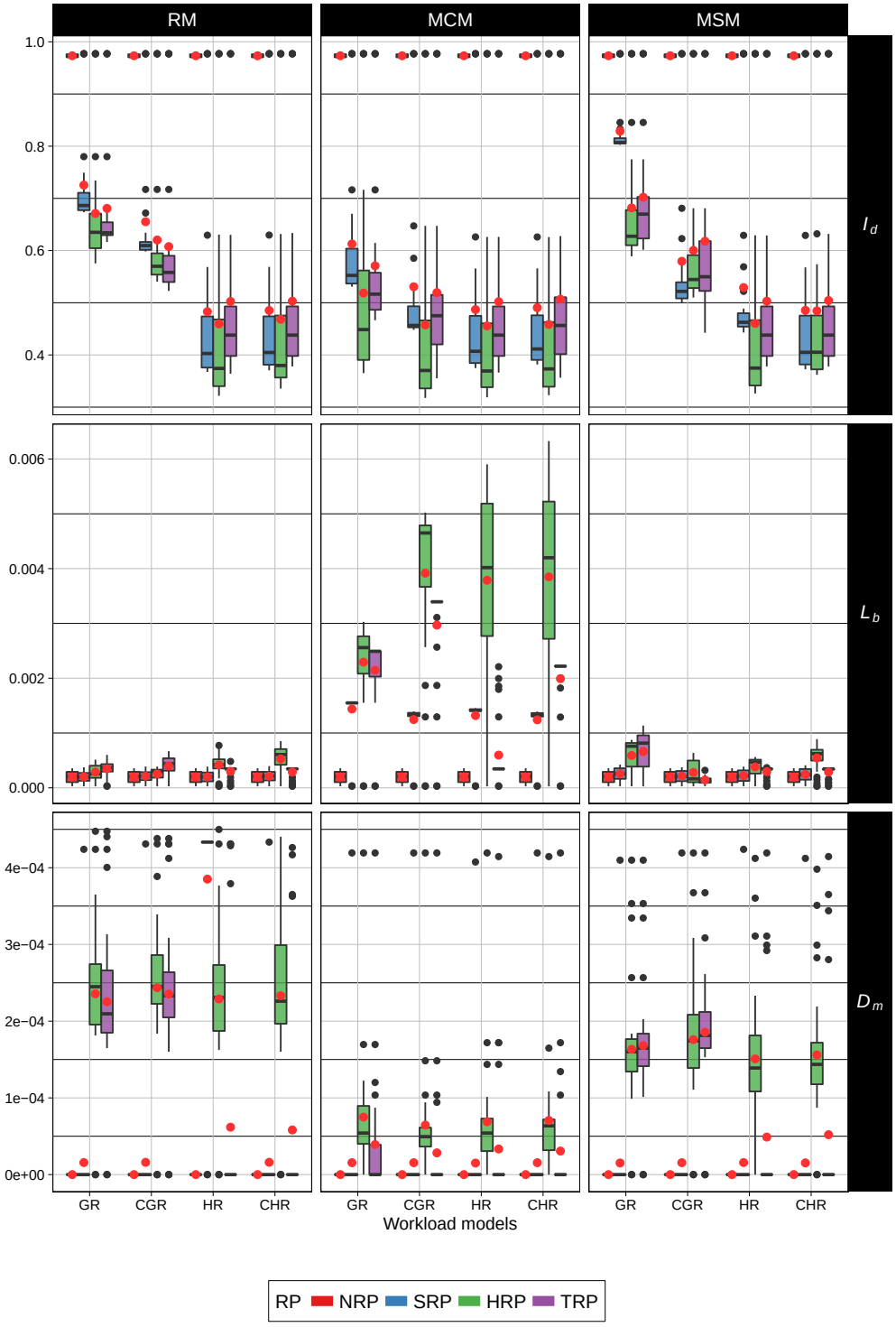


Figure 4.9: Comparison of different incremental repartitioning schemes in a *range-partitioned* database cluster for observing I_d, L_b, and D_m under *Twitter* workload

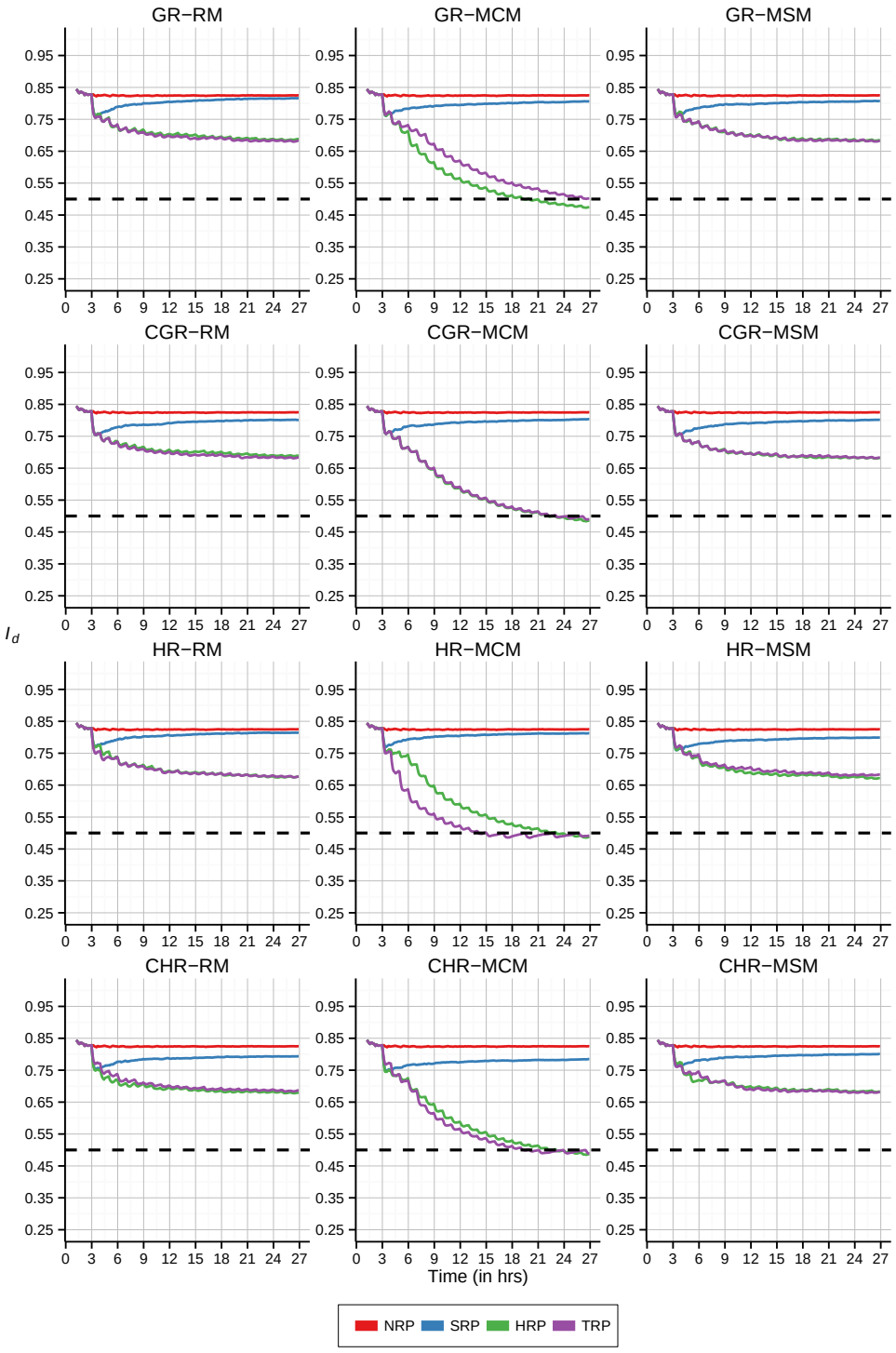


Figure 4.10: Comparison of different incremental repartitioning schemes in a *consistent-hash* partitioned database cluster for observing the variations of I_d under *TPC-C* workload

4.7.1.2 Consistent-Hash Partitioned Database

The overall impacts of DTs is compared in a *consistent-hash partitioned* OLTP database under a *TPC-C* workload as shown in Figure 4.10. As with *NRP*, I_d maintains a steady margin just above 0.8 for all the scenarios, which is higher than a *range-partitioned* system as shown in Figure 4.6. Although *SRP* reduces the margin of I_d between 0.7–0.75 for both the *RM* and *MSM* strategies and between 0.65–0.7 for *MCM*, over time I_d gradually increases to reach the *NRP* margin again. The *RM* and *MSM* strategies perform similarly while the *MSM* has a slightly better result. However, both fail to reduce I_d below the user-defined R_t value for *HRP* and *TRP*. Similar to the *range-partitioned* database, the *MCM* strategy performs well for all of the workload network representations. Although, on average, 15 repartitioning cycles are required for *TRP* compared to the results in Figure 4.6, overall, *MCM* is the only strategy that is capable of holding any preset R_t. *HRP* also performs similar to that in Figure 4.6, and continues to reduce I_d within incremental repartitioning cycles for all workload network representations.

Figure 4.11 compares the I_d variations of different repartitioning schemes for the *Twitter* workload. Similar to the *TPC-C* workload, the I_d margin is steady at round 0.96 for the *consistent-hash partitioned* database under a *Twitter* workload. The results show similar perspectives for the graph and hypergraph (and their compressed forms) based workload network representations that have been already observed in Figure 4.7 for a *TPC-C* workload. For *RM* based data migrations, the *HR* and *CHR* based repartitioning schemes perform much better compared to all the other approaches for a *Twitter* workload. The *GR* and *CGR* based schemes, however, deviate from their initial repartitioning results over time, and maintain a steady margin near 0.75. The *MCM* based approaches, however, succeed in reducing the I_d margin below the preset R_t. The *HRP* and *TRP* schemes for *MCM* gradually reduce I_d between 0.45–0.35 which have been similarly observed for a *TPC-C* workload as well. The *MSM* based schemes for *GR* and *CGR* perform significantly below expectations, and it seems they are not appropriate for handling the social-networking workloads at all. The *HR* and *CHR* based schemes like, *HRP* and *TRP*, perform well using *MSM* based data migrations apart from some fluctuations for *HR*.

While analysing server-level load-balance for the *TPC-C* workload, as shown in Fig-

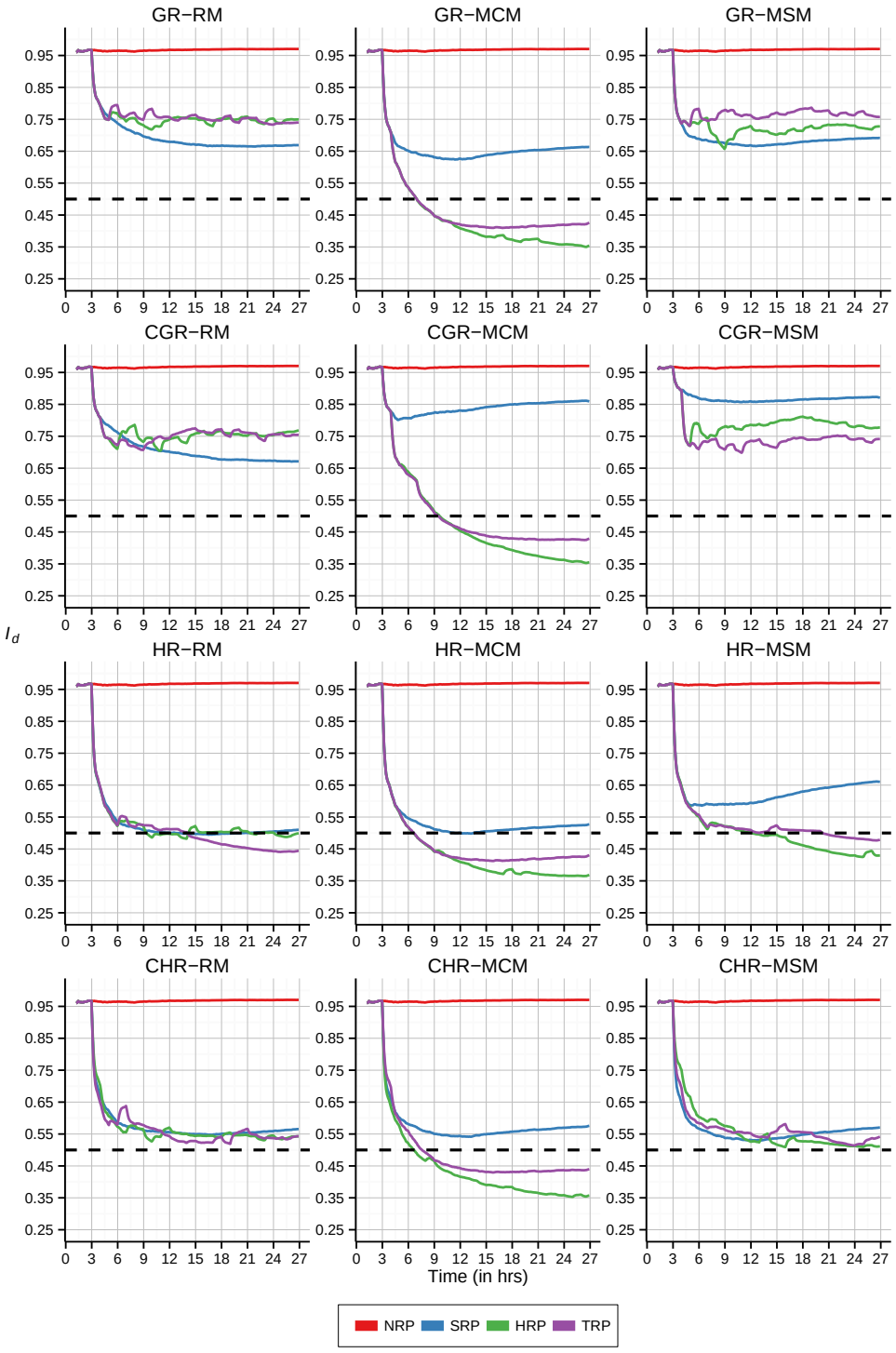


Figure 4.11: Comparison of different incremental repartitioning schemes in a *consistent-hash partitioned* database cluster for observing the variations of I_d under *Twitter* workload

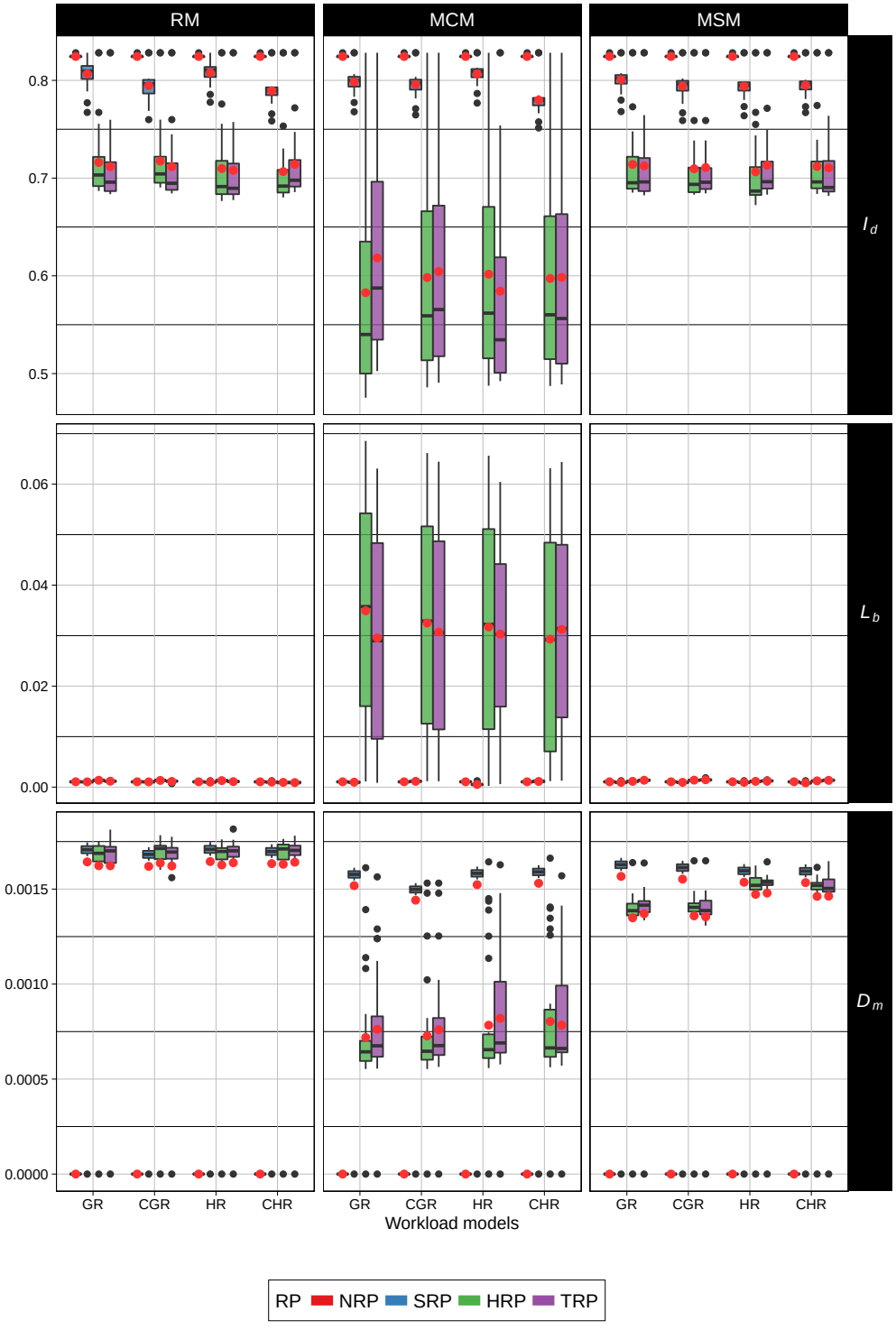


Figure 4.12: Comparison of different incremental repartitioning schemes in a *consistent-hash partitioned* database cluster for observing I_d, L_b, and D_m under *TPC-C* workload

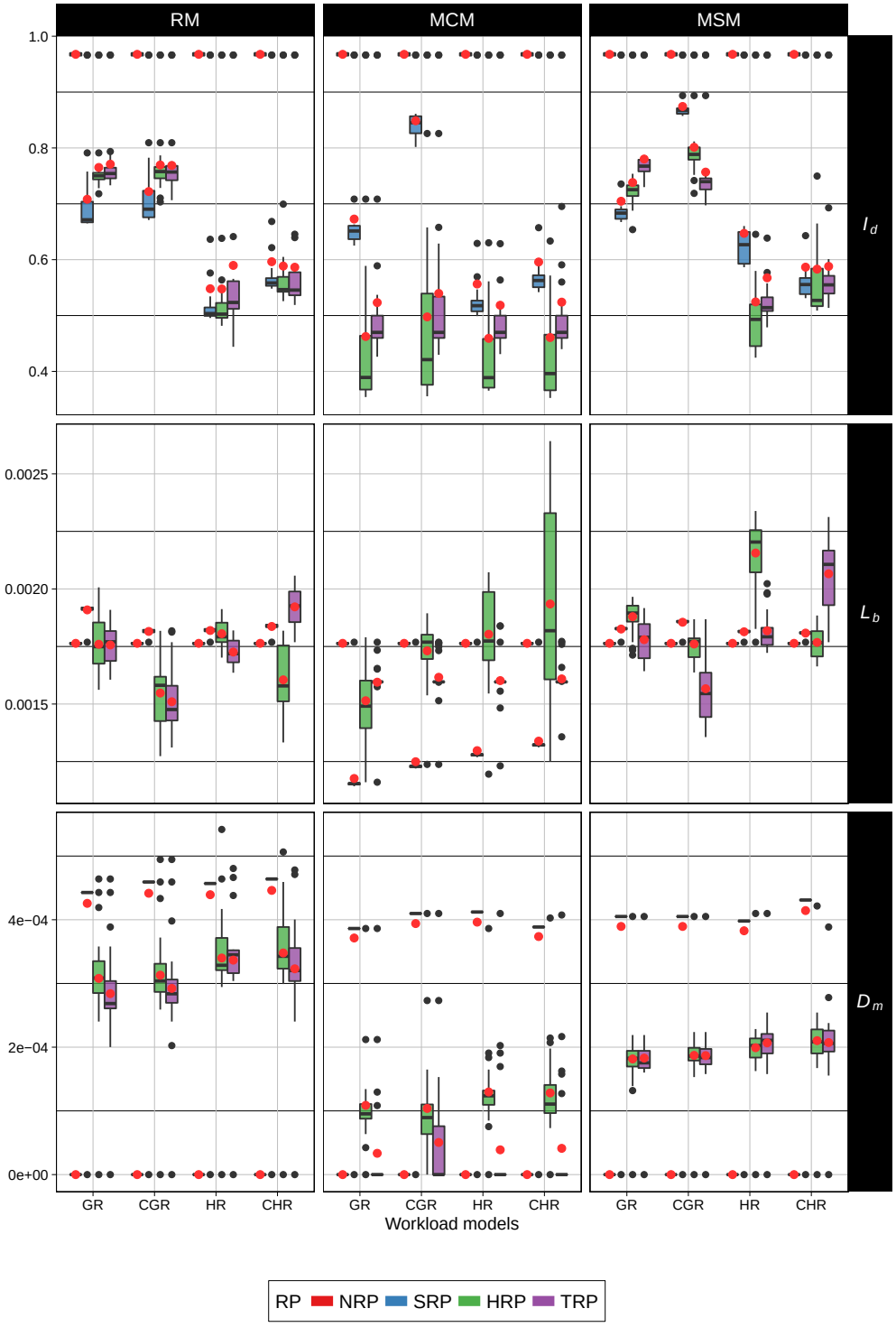


Figure 4.13: Comparison of different incremental repartitioning schemes in a *consistent-hash partitioned* database cluster for observing I_d, L_b, and D_m under *Twitter* workload

ure 4.12, both the *RM* and *MSM* strategies work very strictly in maintaining the load-balance adopted by the initial *consistent-hash* data partitioning in all cases. Although *MCM* strategies increase the flexibility of being *balanced*, *TRP* continues to show better control over load imbalance than the *proactive HRP*. In terms of inter-server data migrations, *MSM* strategies perform marginally better than *RM*; however, both require a large number of data migrations between the DSs to maintain the load-balance. On the other hand, the flexibility adopted by *MCM* sets it apart by performing fewer data migrations; thus, by accepting a slight load imbalance, it can reduce I_d in a controlled way. In comparing the *reactive TRP* with the *proactive HRP* scheme, inter-server data migrations are always higher in the former.

Figure 4.13 presents the load-balance and data migration statistics for the *Twitter* workload using box-plots. Due to the strong load-balance properties of *consistent-hash partitioned* system, the values of L_b are consistent throughout the repartitioning cycles. For *RM*, the *GR* and *CGR* based schemes perform better than the *HR* and *CHR* approaches which are orthogonal in the case of *MCM* based data migration schemes. In the *MCM* strategy, similar observations are found where the *GR* and *CGR* based approaches perform better than the rest. Similar to Figure 4.9, the *RM* and *MSM* strategies require more data migrations than the *MCM* in order to maintain an adequate level of load-balance in the cluster. Overall, the *GR* and *CGR* based techniques require slightly fewer data migrations than the *HR* and *CHR* based repartitioning schemes.

4.7.2 Comments on the Experimental Results

From the above discussion, it is clear that the *TRP* based *MCM* strategy is better suited for both *range-* and *consistent-hash partitioned* databases under the *TPC-C* workload. On the other hand, the *CGR-RM* and *HR-RM*/*CHR-MSM* schemes are better suited for *range-* and *consistent-hash partitioned* databases, respectively, for the *Twitter* workload. Compared with the *proactive* approach, the *reactive* incremental cycles are capable of maintaining a user-defined R_t while performing a constant server-level load-balance with the necessary amount of physical data migrations. In terms of workload network representations, *GR* is somehow ambiguous in representing the actual

transactional interactions, and hence not capable of producing the exact workload network. Therefore, in situations where the database cluster is under stress, the results are not consistent with the expected outcomes. *HR* can represent the exact workload network, but this allows the external graph *min-cut* libraries to produce clusters that require a large number of physical data migrations in order to maintain adequate load-balance. *CGR* and *CHR*, on the other hand, provide less flexibility for the external graph *min-cut* libraries, as they compress inter-related edges and hyperedges together, and thus produce better results in reducing the overall impacts of DTs. Overall, *TRP* based *CHR-MCM* incremental repartitioning shows better control and adaptability in a *reactive* manner for both *range-* and *consistent-hash partitioned* databases under the *TPC-C* and *Twitter* workloads.

4.8 Conclusions

Graph-cut based incremental repartitioning is an indirect way of achieving repartitioning benefits, in terms of minimising the impacts of DTs and load imbalance, by performing a minimum number of data migrations. By abstracting the workload level complexities to a higher dimension, graph theoretic schemes can indirectly improve some of the repartitioning KPIs. However, the overall process varies greatly depending on the workload profile and transactional network characteristics. Therefore, there are no direct ways of forcing any of the repartitioning objectives to minimise the KPIs below a preset threshold of R_t. From experimental evaluations, it is clear that, as *TPC-C* and *Twitter* workloads have different transactional profiles and characteristics, no single combination of workload representation and cluster-to-partition mapping strategy can manage to deal with them. For instance, a compressed hypergraph based workload network representation with *MCM* data migration strategy performs better for *TPC-C* workloads, where transactions are highly concurrent and tuple-level transactional associations are primarily controlled by 'Warehouse' and 'Item' ids. On the other hand, graph and compressed graphs are deemed suitable for *Twitter* workloads, where *RM* and *MSM* data migrations strategies help the *HRP* and *TRP* schemes to minimise the repartitioning KPIs. Based on the the *Twitter* workload implementation (as described

in Chapter 3), most of the transactions involve in retrieving *tweets* for a given popular user that are generated by a fixed number of other users they follow in their social-network. In such situation, the repartitioning results from the simulations are seemed optimistic. In overall, although graph *min-cut* based repartitioning scheme is a popular technique used in existing literatures, until now, it has not been explored deeply enough to understand its implications and effectiveness under different circumstances. In next chapter, we propose a greedy heuristic based repartitioning scheme which removes workload level abstractions altogether and provides more flexibility to the system owners for directly influencing the repartitioning decision to achieve specific balance between multiple repartitioning KPIs. Furthermore, we also discuss open issues left with graph *min-cut* based repartitioning scheme and propose ways to deal with them using a greedy approach.

Chapter 5

Incremental Repartitioning with Greedy Heuristics

In the previous chapter, a graph theoretic incremental repartitioning scheme is proposed that uses higher-level network abstractions and utilises balanced graph *min-cut* clustering to reduce the number of DTs in the workload. Furthermore, two cluster-to-partition mapping techniques are used to aid this scheme to provide sub-optimal repartitioning results for minimising data distribution load imbalance and physical data migrations. However, such a model does not allow the system administrators to implement any preferential policy mix of multiple KPIs in order to obtain an optimal or near-optimal repartitioning outcome. In this chapter, a greedy heuristic based scheme is proposed which avoids network abstractions altogether and directly operates at the transaction-level to derive an optimal repartitioning decision.

5.1 Introduction

Greedy optimisation heuristics are often used when it is computationally challenging to find an optimal solution for a problem with large search space. As a simplistic example, if an optimisation problem exists which is trying to minimise/maximise some function $f(x)$ over domain X, then a greedy heuristic computes the value $f(x)$ locally for all x in the neighbourhood space, and selects the local optima to progress to the next level of solution. Similarly, finding an optimal solution for semi-orthogonal repartitioning KPIs is not trivial. A balanced graph *min-cut* based repartitioning scheme, discussed in the previous chapter, is only able to achieve an optimal repartitioning goal on a single direction (i.e., minimise I_d) unless artificial cluster-to-partition mappings are used to achieve some further level of balance with the other repartitioning KPIs, L_b and D_m. Although the objective to achieve an optimal balance for both I_d, L_b, and

D_m in a partitioned database seems orthogonal, such an approach, which focuses on optimising each possible data migration for a particular DT in hand, along with appropriate heuristics, is entirely feasible. In this chapter, a greedy incremental repartitioning scheme is introduced which aims finding a user-defined balance between multiple repartitioning KPIs over time in order to achieve a near-optimal solution to the repartitioning problem. Theoretical arguments and experimental results jointly confirm that by adopting an exhaustive technique with suitable heuristics, optimal results can be guaranteed in a multi-objective demand-driven repartitioning scenario.

The rest of the chapter is organised as follows. Section 5.2 discusses the motivations, significance, and an overview of the proposed repartitioning scheme. Section 5.3 formally introduces the proposed scheme using detailed mathematical and algorithmic constructions. Section 5.4 discusses the experimental results using the proposed repartitioning scheme for different scenarios and parameter settings. Finally, Section 5.5 comments on the challenges of the proposed scheme and concludes the chapter.

5.2 Overview

In this section, a detailed discussion of the shortcomings of graph theoretic repartitioning scheme is presented. This is followed by the development of, and a proposal for a novel greedy heuristic algorithm for optimal incremental repartitioning.

5.2.1 The Practical Aspects of Graph Theoretic Repartitioning

A graph or hypergraph network is an abstract representation of a transactional workload. Abstractions are generally used to hide underlying complexities of a system. Similarly, the network abstractions used in the previous chapter hide the complexities of transactional workloads and provide much simpler representations. Theoretically, while simpler representations present fewer computational complexity, the solutions can at best be only a rough approximation of the original problem. The higher the level of abstraction, the coarser the level of approximation. As a consequence, when a repartitioning solution is derived from graph based abstract representations (using graph *min-cut* techniques), it may not be perfectly aligned with the solutions that would

have been produced using low level transactional representations. We may then conclude that the graph *min-cut* based repartitioning solutions are not necessarily optimal. The primary motivation of this chapter, therefore, is to develop a repartitioning scheme without resorting to any high level abstraction, where multiple KPIs can be traded-off in order to find an optimal solution by achieving a balance between multiple objectives, i.e., reduce the impacts of DTs and maintain server-level load-balance at a certain proportion. Provisioning user level control to trade-off operational KPIs is highly desirable for any distributed OLTP database in practice.

In this chapter, an incremental repartitioning scheme is derived by completely removing the network abstractions used in the previous chapter, and directly utilising the physical properties of a transactional workload to reduce server spans and load imbalance while restricting the amount of inter-server data migrations. This way, the repartitioning solutions can be genuinely mapped to the root causes of the problems emanating from the transactional workload. Now, it is essential to ascertain whether it would even be possible to work at such a low level where no abstraction layer is present. In the proposed approach, all the DTs (that is T_d) within the most recent workload observation window $\mathcal{W}$ are exhaustively examined in a greedy ordering without any backtracking to determine a near-optimal repartitioning solution. Therefore, it is necessary to understand whether such an exhaustive greedy approach can be completed within a polynomial time bound.

The experimental results on graph theoretic repartitioning schemes presented in the previous chapter have been obtained by restricting the repetition pool for unique transaction generations as described in Chapter 3. This is done to avoid any undue influence of external graph clustering libraries, while comparing different repartitioning schemes. In reality, however, the repetitive nature of OLTP workloads is not bounded to a fixed set of unique transactions over an observation window. Therefore, the actual repartitioning time for any graph *min-cut* based scheme will be much higher, and in some cases may be impractical when rapid repartitioning decision making is necessary.

Table 5.1: A comparison of mean repartitioning times using both restricted and unrestricted unique transaction generation models for graph theoretic repartitioning scheme with different workload representations following *MCM* cluster-to-partition mapping strategy

Workload types	Initial data partitioning	Repartitioning time (in seconds)							
		GR		CGR		HR		CHR	
		Restricted	Unrestricted	Restricted	Unrestricted	Restricted	Unrestricted	Restricted	Unrestricted
TPC-C	Range	1.456	10.442	1.218	5.684	5.092	44.399	10.479	129.363
	Consistent-hash	0.930	11.630	0.934	6.118	7.480	57.598	16.458	187.890
Twitter	Range	0.303	0.960	0.437	0.943	0.666	0.908	0.536	1.314
	Consistent-hash	0.418	0.858	0.390	0.740	1.069	1.639	1.126	1.412

Table 5.1 presents a comparison between average repartitioning times[1] for the graph *min-cut* based scheme following MCM cluster-to-partition mapping strategy, using both restricted and unrestricted unique transaction generation models. From the observations, it is clear that the average repartitioning time for unrestricted unique transaction generations is several magnitudes higher than the restricted model, for all workload types and representations.

For real-life production deployment of very large-scale OLTP databases, the size of the observed workload network can grow beyond millions of vertices and edges. It is well known that the order of graph theoretic algorithms in general is $O(|\mathcal{V}|+|E|)$ where $\mathcal{V}$ and E are the set of vertices and edges of the underlying graph, respectively. Therefore, in practical scenarios, incremental repartitioning following a graph theoretic approach can be problematic, when an on-the-spot repartitioning decision is required. Furthermore, with a sub-optimal outcome, the repartitioning process may also fail to make the right balance within the KPIs for multi-objective repartition goals.

5.2.2 Challenges in Finding an Optimal Repartitioning Decision

Based on the discussion on the properties of transactional workloads in Chapter 3, OLTP transactions usually operate over a fixed number of data tuples within a database in a repetitive manner. For example, *TPC-C* transactions are mostly controlled by the given *Warehouse* 'id' which associates its related data tuples in the database. Similarly, *Twitter* transactions modelled in *OLTPDBSim* are primarily steered by the given *User* and *Tweet* ids that eventually control the number of *Tweets* to be retrieved from the *Followers* and *Follows* relationships.

OLTP workloads, in a sense, operate over a network of relationships. Thus, the size of the dataset for individual transactions are constant for a particular class or type of transaction from the transaction profile of an OLTP application. Therefore, it can be assumed that any particular tuple is shared within a relatively small number of DTs. Furthermore, from the study of network science it is found that such information access patterns in the Web do not follow the characteristics of a *random* network, rather the

[1] Repartitioning times here include both graph compression and repartitioning decision making times, while excluding the time taken for physical data migrations.

repetitive visits by the users and their information retrieval patterns follow the *scale-free* property [158–161].

Consider the scenarios involving individual banking transactions, as an example. Data tuple representing individual account numbers can present in the transactions initiated by the particular account holder. Considering that the number of transactions initiated by individuals is limited and the people they conduct these transactions with are also limited, it can be safely assumed that these numbers will remain relatively constant, even if the number of transactions in the overall banking application grows over time. Similarly, in a social networking application, it is possible to show that every user is connected with one another despite having a limited number of individual connections. Therefore, the degree[2] of a data tuple in the workload network representing the number of its incident transactions may be assumed to be a small constant. This claim can be supported from studies in the existing literature. For instance, the average distance between all pairs of users within Facebook was 5.28 hops in 2008 and became 4.74 by 2011, as the global network became more and more connected [162]. Similar characteristics can also be found in other OLTP applications. Furthermore, this phenomenon, which was originally observed from the studies of *Small-World Problem* [163, 164], has been extensively studied in the literature.

5.2.3 Proposed Greedy Repartitioning Scheme

The main concept of the proposed greedy heuristic as well as some implementation issues are presented in the following two subsections.

5.2.3.1 Main Concept

Let us consider the sample OLTP database and transactional workload in Tables 2.4 and 4.1, respectively, to assist in presenting the main concept of the proposed greedy heuristic based repartitioning scheme. Let us further assume that these seven unique transactions occur in the workload with frequencies 10, 5, 3, 2, 2, 1, and 1, in that order. Note that the server span of three DTs is 2 and rest are 1. The impacts of

[2] The *degree* of a vertex in a network represents the number of edges it has to other vertices.

distributed transactions and server level load balancing KPIs of the system can be calculated using (3.9) and (3.12), respectively, as

$$I_d = \frac{2\times10+2\times5+1\times3+1\times2+2\times2+1\times1+1\times1}{2\times10+2\times5+2\times3+2\times2+2\times2+2\times1+2\times1} = \frac{41}{48},$$
$$L_b = \frac{\sqrt{\frac{(10-10)^2+(10-10)^2}{2}}}{10\sqrt{2-1}} = 0.$$

Now, the server span of a DT can be reduced by 1 if all its data tuples in a physical server are moved to another server from where the DT uses some data tuples. For example, server span of DT τ_2 can be reduced to 1 by migrating either tuples $\{4,6\}$ from server S_1 to S_2 or tuples $\{1,9,11\}$ from S_2 to S_1. The first data migration plan does not change the span of any other transaction; but the load of servers changes by 2. Effectively, the KPIs are changed as follows:

$$I'_d = \frac{2\times10+1\times5+1\times3+1\times2+2\times2+1\times1+1\times1}{2\times10+2\times5+2\times3+2\times2+2\times2+2\times1+2\times1} = \frac{36}{48},$$
$$L'_b = \frac{\sqrt{\frac{(8-10)^2+(12-10)^2}{2}}}{10\sqrt{2-1}} = \frac{1}{5}.$$

The second plan increases the span of MNDTs τ_3 and τ_4 by 1, and the load of servers is changed by 3. Effectively, the KPIs are changed as follows:

$$I''_d = \frac{2\times10+1\times5+2\times3+2\times2+2\times2+1\times1+1\times1}{2\times10+2\times5+2\times3+2\times2+2\times2+2\times1+2\times1} = \frac{41}{48},$$
$$L''_b = \frac{\sqrt{\frac{(13-10)^2+(7-10)^2}{2}}}{10\sqrt{2-1}} = \frac{3}{10}.$$

It is, therefore, possible to measure the net improvement (NI) of a KPI (ΔKPI) for any migration plan. For the first plan above, $\Delta' I_d = I_d - I'_d = \frac{5}{48}$ and $\Delta' L_b = L_b - L'b = -\frac{1}{5}$. Similarly, for the second plan above, $\Delta'' I_d = I_d - I''_d = 0$ and $\Delta'' L_b = L_b - L''_b = -\frac{3}{10}$. Note that the higher these values the higher the improvement and that negative values indicate deterioration.

By considering ΔKPI per data tuple migration, it is possible to compare feasible migration plans for a DT to find the best one. For example, the first plan above is best for I_d improvement as $\frac{\Delta' I_d}{2}(=\frac{5}{96}) > \frac{\Delta'' I_d}{3}(=0)$ and both plans are best for L_b improvement

as $\frac{\Delta' L_b}{2} = \frac{\Delta'' L_b}{3} = -\frac{1}{10}$. Using the same principle, DTs can also be compared to find the best one. This ultimately establishes a greedy heuristic for incremental repartitioning where iteratively the best migration plan of the best DT is executed. Note that due to using per data improvement, the greedy heuristic also indirectly keeps the inter-server data migrations KPI D_m in check.

This greedy approach is flexible enough to optimise the repartitioning problem subject to the weighted mean KPI $\lambda I_d + (1-\lambda)L_b$ where λ is a user-defined weight in the range of $[0,1]$. In that case, KPI is obtained using the same weighted mean formula as

$$\begin{aligned}\Delta' KPI &= (\lambda I_d + (1-\lambda)L_b) - (\lambda I'_d + (1-\lambda)L'_b) = \lambda(I_d - I'_d) + (1-\lambda)(L_b - L'_b) \\ &= \lambda \Delta' I_d + (1-\lambda)\Delta' L_b\end{aligned}$$

The example used in this section has the opportunity to reduce the span of a DT by 1 only as $|S| = 2$. For large OLTP applications with more than 2 physical servers, the concept can be extended to consider migration plans that move data tuples from multiple servers into another server, and hence reduce the span of a DT by more than 1. Such generalisation can be bounded to γ-span, $1 \leq \gamma < |S|$, where migration plans are restricted such that data tuples from most servers are moved.

Finally, another strength of the proposed greedy approach is its ability to terminate a repartitioning initiative as soon as the user-defined target is achieved as migration plans are selected one-by-one. The graph theoretic approach proposed in the previous chapter cannot be terminated in the middle as the migration plans are selected together only after the min-cut clustering is completed.

5.2.3.2 Implementation Issues

To implement such a multi-objective optimisation algorithm, consider a *priority queue* $\mathcal{PQ}$, which orders the DTs in T by their maximum gains in reducing the adverse impacts of the repartitioning problem, i.e., impacts of DTs (I_d) and load imbalance. Transactions are *peeked*[3] from $\mathcal{PQ}$ for processing, i.e., redistributing its tuple set to achieve the repartitioning objectives. The set of incident transactions is also updated

[3] *Peek* retrieves, but does not remove, the highest priority element from the head of the queue.

accordingly and reinserted back in $\mathcal{PQ}$ to reorder the priority. This entire process affects the repartitioning gains achievable by its incident transactions in future. Either the gain achieved by the transaction under processing will suppress the future gains by its incident transactions, or vice versa. Or, the current achievement may be subjugated by the changes made from processing its incident transaction in future. Therefore, an already processed transaction may need to reappear again and again in $\mathcal{PQ}$, hence creating *ping-pong effect* which makes it an *NP-complete* problem. This process continues until $\mathcal{PQ}$ becomes empty.

In regard to the computational order of the above algorithm, it takes $\log n$ time to *poll* [4] a single element from a $\mathcal{PQ}$, and reorder it as described above. Let, c be the constant number of incident DTs for any DT in $\mathcal{PQ}$. Therefore, it will take $(c+1)\log n$ time to poll each DT from $\mathcal{PQ}$, and then reorder the it accordingly. For n number of DTs in $\mathcal{PQ}$, it will take $(c+1)n\log n$ time to complete a single iteration of the incremental repartitioning process. Despite the fact that the constant c can be a large value depending on the workload characteristics, the complexity of this algorithm can be reduced to $O(n\log n)$ which is polynomial. However, processing each transaction involves processing its incident set of transactions and processing any of these incident transaction may require modifying the current transaction under processing. This phenomenon may create a *ping-pong effect* within the workload network which does not have any polynomial bound to finish, hence the algorithm outlined above may not be able to complete in polynomial time as expected. This proves *NP-completeness* of the above mentioned approach, therefore no optimal solutions can be found to solve this problem.

To find an algorithm for the above problem which completes in polynomial time requires the development of special heuristics. As discussed in Chapter 4, balanced graph *min-cut* based incremental repartitioning schemes also use heuristics where workload network abstractions are created using graph, hypergraph, or their compressed representations. Experimental results from Chapter 4 have also shown that the solutions are not optimal in achieving specific repartitioning objectives such as minimising the impacts of DT or maintaining server-level load-balance. However, workload network

[4] *Poll* – retrieves and remove, the highest priority element from the head of the queue.

abstractions that are created at far higher level do not provide sufficient flexibility to manipulate individual transactions at the lowest level. Therefore, in this chapter simple heuristics are used in incremental repartitioning schemes which simultaneously allow them to operate over individual DTs while exhaustively processing the $\mathcal{PQ}$.

A simple heuristic to avoid creating a *ping-pong* effect in processing the incident transactions within $\mathcal{PQ}$ is to forbid processing a DT which may adversely affect the gains achieved by any of its incident transactions (i.e., predecessors) which have been already processed. As $\mathcal{PQ}$ is ordered according to the maximum gain achievable by each transaction, therefore bypassing the processing of a future incident transaction of a transaction that has been already processed to achieve a higher gain has a neutral affect on the overall impacts of DT within the system. This simple heuristic allows the algorithm to complete in polynomial time with complexity $O(n \log n)$ as the number of elements in $\mathcal{PQ}$ reduces at each iteration. In the following sections, the above greedy heuristic based repartitioning algorithm will be presented using mathematical notations based on the above heuristics and then evaluated in simulation to show their effectiveness in achieving particular repartitioning goals using the more realistic unrestricted transaction generation model.

5.3 Greedy Heuristic based Incremental Repartitioning Scheme

Let T be the set of all transactions. Let $\Delta_i = \{\delta_{i,1}, ..., \delta_{i,|\Delta_i|}\}$ denote the tuple set of transaction τ_{d_i}, and D_{S_j} be the tuples residing in server S_j. Let $\psi(\delta)$ denote the resident server of tuple δ and $\psi(\Delta) = \cup_{\delta \in \Delta} \psi(\delta)$ denote the residing set of servers for all tuples in Δ. Note that $\psi(\Delta_i)$ denotes the set of servers spanned by τ_{d_i}. Let $\Delta_{i,j} = \{\delta \in \Delta_i \big| \psi(\delta) = S_j\}$ denote the tuple set of τ_{d_i} stored in server S_j and $\Delta_{i,A} = \cup_{j \in A} \Delta_{i,j}$ denote the collective tuple set of τ_{d_i} stored in the set of server $A \subset \psi(\Delta_i)$. Let $\tilde{\Upsilon}_\tau$ be the expected period of recurrence of transaction τ. Let $T_i^{\bowtie} = \{\tau_{d_j} \in T_d \setminus \{\tau_{d_i}\} \big| \Delta_j \cap \Delta_i \neq \phi\}$ be the set of DTs incident to τ_{d_i}. Let $m_{i,A \to b}$ denote a migration plan for τ_{d_i} by moving tuple set $\Delta_{i,A}$ from A to a single server $b \in \psi(\Delta_i) \setminus A$.

Let $\Delta I_d(m_{i,A \to b})$ and $\Delta L_b(m_{i,A \to b})$ denote the net improvement per data migration

(NIPDM), the higher the better, in I_d and L_b, respectively, that are calculated as follows:

$$\Delta I_d(m_{i,A\to b}) = \frac{I_d - I_d[\![m_{i,A\to b}]\!]}{|\Delta_{i,A}|} = \frac{1}{|\Delta_{i,A}|} \frac{\frac{|A|}{\tilde{\Upsilon}_{\tau_{d_i}}} + \sum_{\tau_{d_j}\in T_i^{\bowtie}} \frac{|A|-|\psi(\Delta_{j,A}\setminus\Delta_{i,A})|-F_{0/1}(j,b)}{\tilde{\Upsilon}_{\tau_{d_j}}}}{\sum_{\tau\in T}\frac{|S|}{\tilde{\Upsilon}_{\tau}}} \tag{5.1}$$

where

$$F_{0/1}(j,b) = \begin{cases} 0, & b \in \psi(\Delta_j); \\ 1, & \text{otherwise} \end{cases}$$

and

$$\Delta L_b(m_{i,A\to b}) = \frac{L_b - L_b[\![m_{i,A\to b}]\!]}{|\Delta_{i,A}|} = \frac{1}{|\Delta_{i,A}|} \frac{\sigma_{\mathcal{D}_S} - \sigma_{\mathcal{D}_S}[\![m_{i,A\to b}]\!]}{\mu_{\mathcal{D}_S}\sqrt{|S|-1}} \tag{5.2}$$

where the notation $\blacksquare[\![m_{i,A\to b}]\!]$ is used to refer to a measure after the migration plan $m_{i,A\to b}$ is executed.

Let $\kappa_\lambda(m_{i,A\to b})$ be the weighted mean NIPDM (WMNIPDM) for migration plan $m_{i,A\to b}$ with weight λ, $0 \le \lambda \le 1$, defined as

$$\kappa_\lambda(m_{i,A\to b}) = \lambda\Delta I_d(m_{i,A\to b}) + (1-\lambda)\Delta L_b(m_{i,A\to b}). \tag{5.3}$$

For already executed migration plans covering set of transactions T_{ex}, let $\widetilde{M}_{i,T_{ex}}$ represents the set of *non-contradicting* migration plans for τ_{d_i} such that, if executed, none of the plans would extend the server span of any of the transactions in T_{ex}.

Given λ, T_{ex}, and γ-span, an *upper limit* on the cardinality of A, let $m_{i,A\to b}^{\gamma,\lambda,T_{ex}}$ be the best non-contradicting migration plan, achieving the maximum possible WMNIPDM, for τ_{d_i}, which is defined as

$$m_{i,A\to b}^{\gamma,\lambda,T_{ex}} = \begin{cases} \textbf{null}, & \widetilde{M}_{i,T_{ex}} = \phi; \\ \underset{\forall m_{i,A\to b}\in\widetilde{M}_{i,T_{ex}}\,\big||A|\le\gamma}{\operatorname{argmax}} \kappa_\lambda(m_{i,A\to b}), & \text{otherwise.} \end{cases} \tag{5.4}$$

Finally, Algorithm 5.1 lists the greedy heuristic based incremental repartitioning algorithm as follows.

Algorithm 5.1 The greedy heuristic based incremental repartitioning algorithm

1: **procedure** PROCESSPQ(S, T_d, γ, λ)
2: $T_{ex} \leftarrow \phi$
3: **while** $I_d > th$ **do**
4: $i^* = \underset{\forall i}{\text{argmax}}\ m_{i,A\rightarrow b}^{\gamma,\lambda,T_{ex}}$
5: **if** $m_{i^*,A\rightarrow b} =$ **null then**
6: EXIT
7: **end if**
8: Execute migration plan $m_{i^*,A\rightarrow b}$
9: $T_{ex} \leftarrow T_{ex} \cup \{\tau_{d_{i^*}}\}$
10: **end while**
11: **end procedure**

5.4 Experimental Evaluation

In this section, we evaluate the performance of the proposed greedy heuristic based repartitioning algorithm in its four different variations. Note that, in this proposed approach for each DT in hand, we calculate the NIPDM values for I_d and L_b for each possible migration plan according to (5.1) and (5.2), respectively. These are the per data migration gains for I_d and L_b. Finally, (5.3) provides the corresponding $m_{i,A\rightarrow b}$'s WMNIPDM depending on the pre-defined weight factor λ. For individual $m_{i,A\rightarrow b}$, it is possible to calculate the I_d and L_b gains in (5.1) and (5.2) without dividing by $|\Delta_{i,A}|$, i.e., the required amount of data to be migrated from A. This will provide the best I_d and L_b gain values for the corresponding $m_{i,A\rightarrow b}$ without considering any data migration optimisation. Furthermore, by relaxing or restricting the $|A| \leq \gamma$ condition in (5.4), we can also control the possible number of migration plans to be generated and computed for WMNIPDMs. By restricting the condition as $|A| = \gamma$ in (5.4), we can expedite the convergences of the I_d and L_b KPIs by allowing only the largest possible many-to-one data migration plans to be computed.

By controlling the behaviour of whether data migration optimisation is applied or not, and whether fast or slow convergence is required, it is possible to simulate several variations of the proposed algorithm. This will allow us to examine the adaptability of the proposed greedy heuristic based incremental repartitioning scheme for different scenarios of OLTP applications. Based on these two criteria, we examine four different variations of the proposed algorithm as

(1) *with Division no Restriction (wDnR)*—which is the original version of the proposed algorithm that calculates the NIPDM values of I_d and L_b for all possible migration plans of an individual DT.

(2) *with Division with Restriction (wDwR)*—which restricts the number of possible migration plans to be considered by imposing $|A| = \gamma$ condition while calculating the NIPMD values of I_d and L_b for an individual DT.

(3) *no Division no Restriction (nDnR)*—which will allow the migration plans with the highest I_d and L_b net improvements to be selected for an individual DT

(4) *no Division with Restriction (nDwR)*—which imposes the $|A| = \gamma$ restriction on selecting the possible data migration plans while ensuring the highest I_d and L_b gains to be computed for an individual DT.

The greedy heuristic based repartitioning algorithm is evaluated using *OLTPDBSim* with the same simulation parameters listed in Table 4.2 using an unrestricted transaction generation model and a pre-defined repartitioning triggering threshold R_t of 0.5 for four DSs. In order to further examine the individual variation of the proposed algorithm, we vary λ from 0.0 to 1.0 with an interval of 0.25, prioritising L_b and I_d gains, respectively, during the best migration plan selection for an individual DT. These wide-scale experimentations show the proposed repartitioning scheme's capability in handling different scenarios of modern OLTP applications.

5.4.1 Analysis of Experimental Results for *TPC-C* and *Twitter* Workloads

In this section, experimental results are shown for a *range-* or *consistent-hash partitioned* database cluster under *TPC-C* and *Twitter* workloads, respectively.

5.4.1.1 Range-Partitioned Database

Figure 5.1 presents the experimental results using a *range-partitioned* database under *TPC-C* workload. From Figure 5.1(a), one can visualise the difference in I_d convergences of the four variations of the proposed incremental repartitioning scheme for

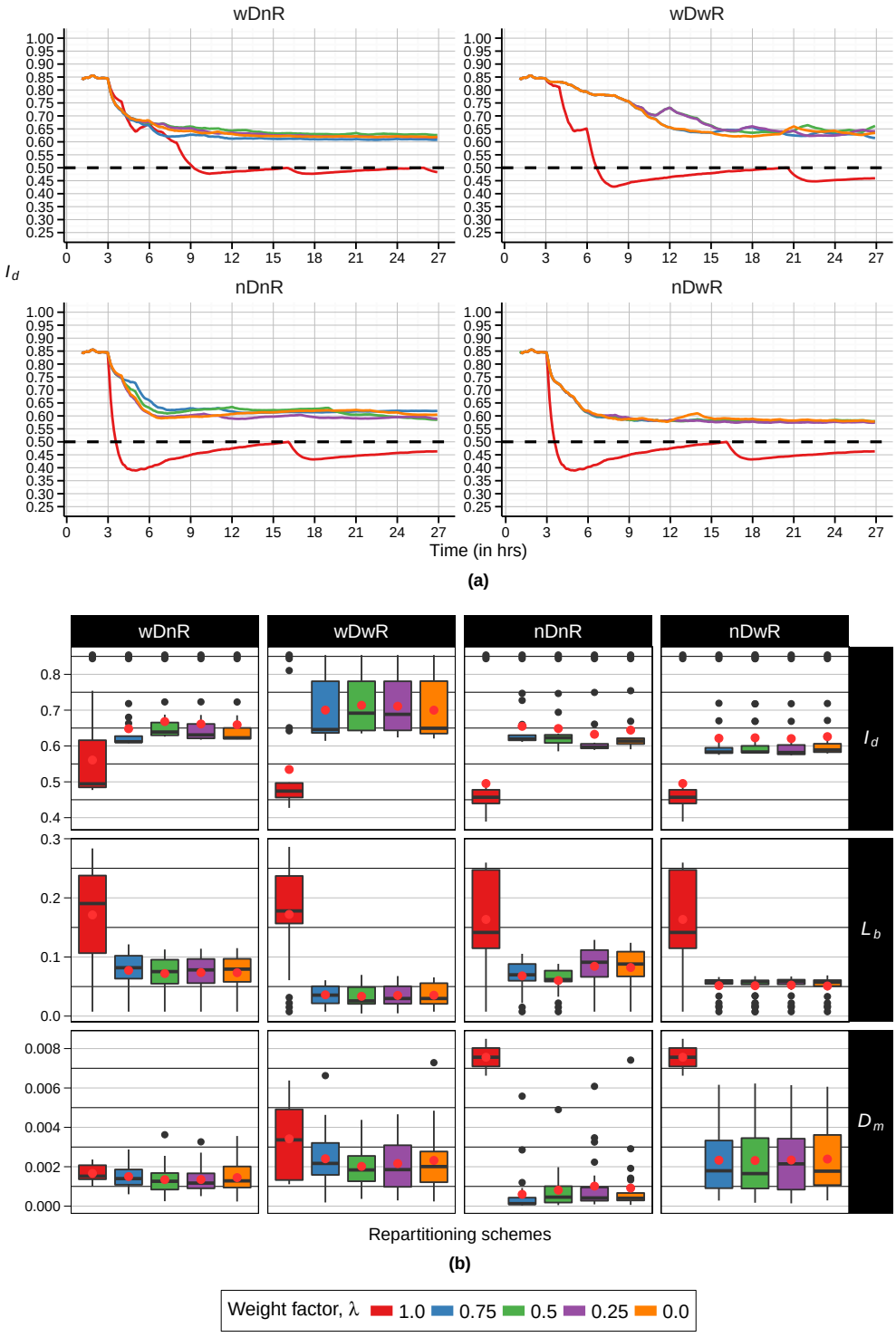


Figure 5.1: Comparison of greedy heuristic based repartitioning schemes with different λ values in a *range-partitioned* database cluster for observing I_d, L_b, and D_m under *TPC-C* workload

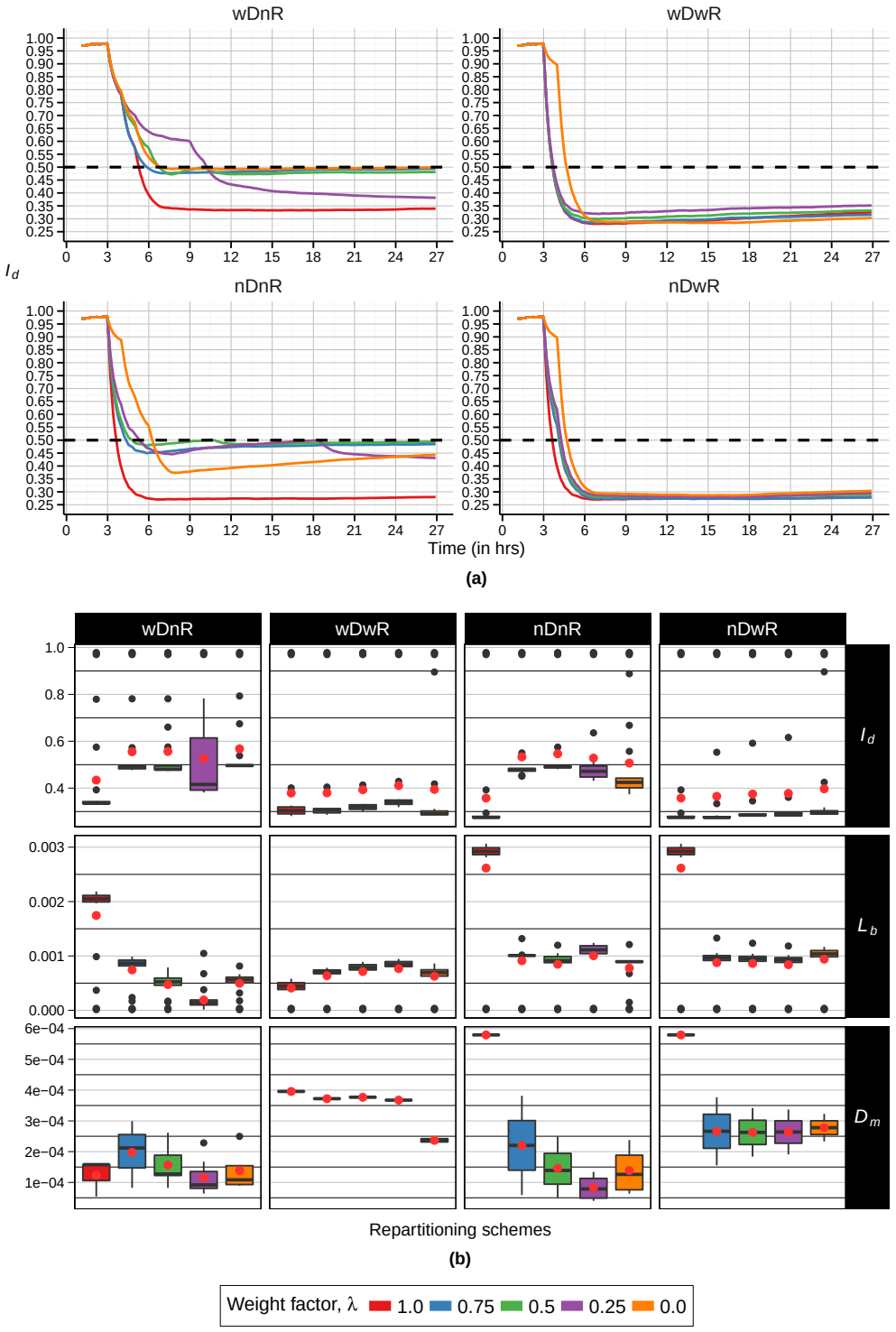


Figure 5.2: Comparison of greedy heuristic based repartitioning schemes with different λ values in a *range-partitioned* database cluster for observing I_d, L_b, and D_m under a *Twitter* workload

$\lambda = 1.0$. The *wDnR* is the exact implementation of the proposed algorithm which converges slowly to reduce I_d below R_t. The convergence of *wDwR* is faster the former one as only the migration plans with $\gamma = |S| - 1$ are selected. For both *nDnR* and *nDwR* variations, the I_d convergences for $\lambda = 1.0$ are exactly the same. This is because both of these select the best migration plans prioritising I_d gain which is independent of γ condition when no data migration optimisation is used. Notice that, for λ values 0.0 to 0.75, the resulting I_d performances are very close to each other, which indicates that the relationship between I_d and L_b is not fully orthogonal. Therefore, it is not possible to achieve a situation where I_d can be entirely sacrificed in cases for $\lambda < 1.0$.

This phenomenon is more clearly captured in the results shown in Figure 5.1(b), where the I_d, L_b, and D_m metrics are presented in box plots. The semi-orthogonal relationships between I_d and L_b as well as L_b and D_m can be observed with different λ values. For all situations where $\lambda < 1.0$ there are hardly any differences in L_b and D_m KPIs for all of the variations examined.

For *Twitter* workload using the *range-partitioned* database as shown in Figure 5.2, the results are rather difficult to distinguish for $\lambda = 1.0$ in all of the variations. Due to the specific characteristics of a *Twitter* workload, all the variants of the proposed algorithm have successfully managed to reduce the overall impacts of DT under the preset threshold as can be seen from Figure 5.2(a). The only distinctions to be noticed are for the *wDnR* and *nDnR* variations where the I_d lines are rather separable due to the fact that different migration plans are selected for different λ values as no restriction is applied.

Figure 5.2(b) presents I_d, L_b, and D_m KPIs in box plots observed for the entire simulation period. In the cases of *wDnR* and *wDwR*, the L_b and D_m performances are very similar for cases where $\lambda < 1.0$. For the *nDnR* and *nDwR* variations, the reduction of I_d is achieved in $\lambda = 1.0$ by performing more data migrations which also leads to high L_b values.

5.4.1.2 Consistent-Hash Partitioned Database

Figure 5.3 presents the results obtained from experimenting in a *consistent-hash partitioned* database under a *TPC-C* workload. As expected, the results in Figures 5.3(a) and (b)

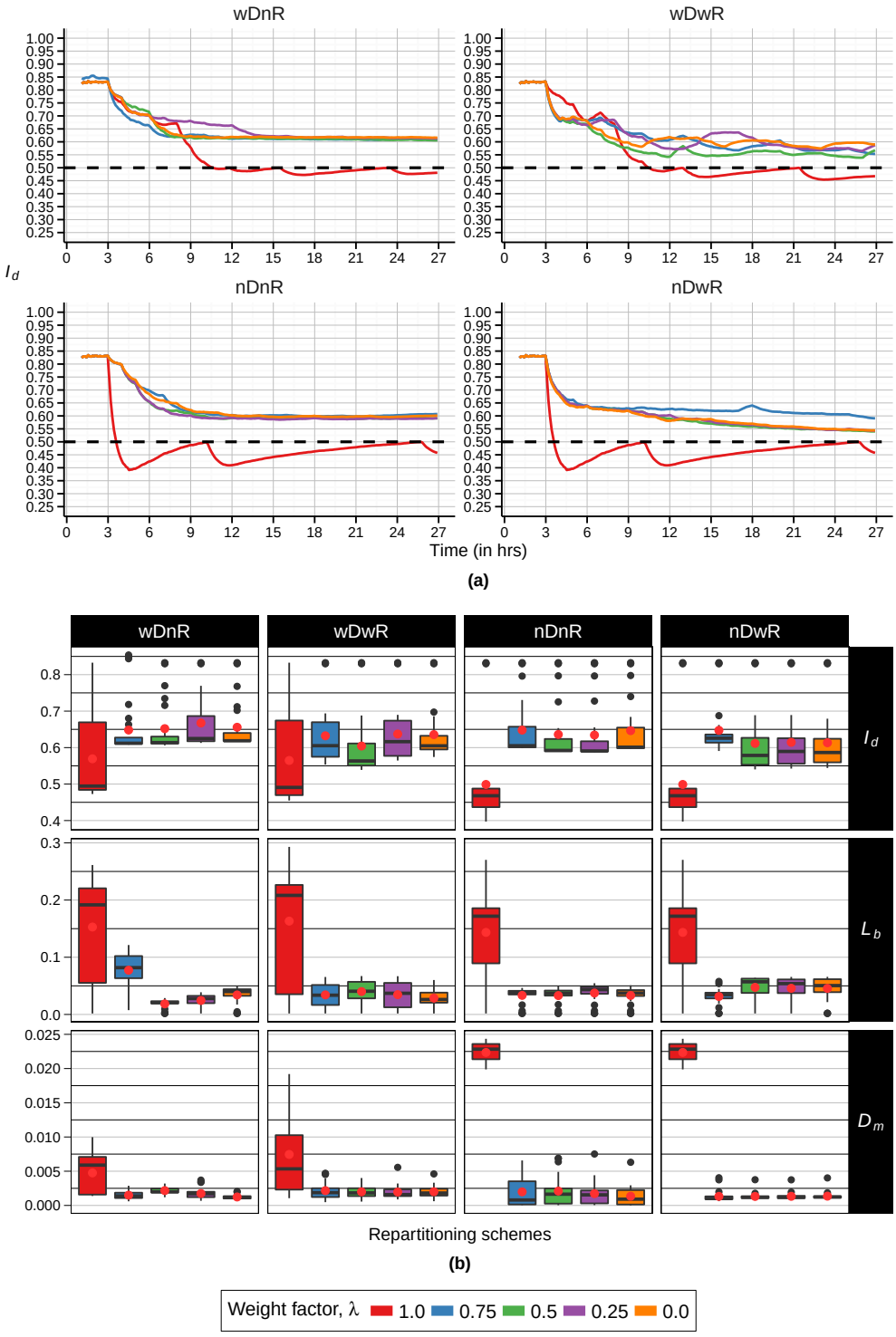


Figure 5.3: Comparison of greedy heuristic based repartitioning schemes with different λ values in a *consistent-hash partitioned* database cluster for observing I_d, L_b, and D_m under a *TPC-C* workload

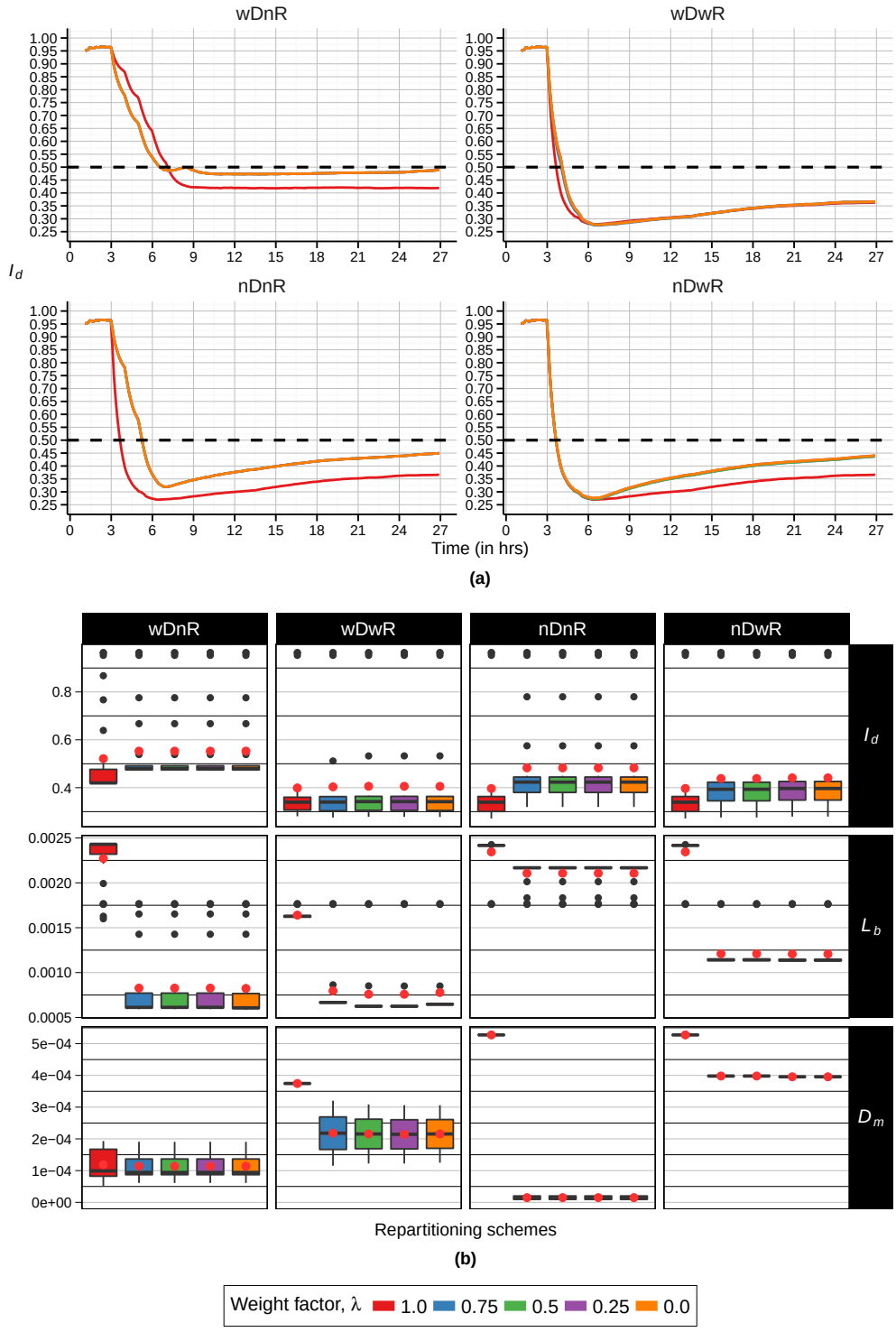


Figure 5.4: Comparison of greedy heuristic based repartitioning schemes with different λ values in a *consistent-hash partitioned* database cluster for observing I_d, L_b, and D_m under a *Twitter* workload

reveal close similarities with the results shown in Figure 5.1. For the *wDnR* and *wDwR* variations, the slow convergences of I_d for $\lambda = 1.0$ is clearly visible. For all other cases ($\lambda < 1.0$), the variations of I_d lines in *wDnR* flatten out and are almost inseparable, while for the added restriction in data migration plan generation the I_d performances are within close range. Figure 5.3(b), where the resultant KPIs are presented using box plots, shows similar characteristics to those observed for *range-partitioned* database cases in Figure 5.1(b). The L_b and D_m KPI performances are very similar to each other in all cases except for $\lambda = 1.0$. For the *nDnR* and *nDwR* variations, the I_d fast convergences are achieved by performing more data migrations in bursts. On the other hand, due to the optimised data migrations over consecutive repartitioning cycles, the *wDnR* and *wDwR* variations slowly converge I_d towards R_t.

The experimental results using a *consistent-hash partitioned* database under a *Twitter* workload are presented in Figure 5.4. Again, these results are very similar to the results obtained in Figure 5.2. The I_d line graphs, as shown in Figure 5.4(a), for each variant of the proposed algorithm are very close to each other and are due to specific characteristics of the *Twitter* workload. Figure 5.4(b) shows the I_d, L_b, and D_m values for the experimental variants in box plots which again affirm similarity with those in Figure 5.2.

5.4.2 Combined Analysis of Experimental Results for $\lambda = 1.0$

In this section, experimental results for all of the abovementioned four variations where $\lambda = 1.0$ are observed together and compared against *SWORD* [93, 124, 125]—an exhaustive incremental repartitioning scheme discussed in Section 2.6.4.2.4. From Figure 5.5(a) for a *TPC-C* workload, it is clear that, due to the exhaustive search for suitable swapping pairs of virtual nodes which in most cases are limited in number, the value of I_d flatten out at 0.8 and is unable to converge towards the R_t threshold of 0.5. For a *Twitter* workload, although *SWORD* manages to aggressively reduce I_d below R_t following that it gradually starts to show signs of divergence from the preset R_t margin. To confirm these for both the *TPC-C* and *Twitter* workloads, we ran the simulations for 72 h, and observe that *SWORD* is unable to maintain an effective I_d level for both cases. Figure 5.5(b) also confirms the high volume of data migrations that *SWORD* needs to

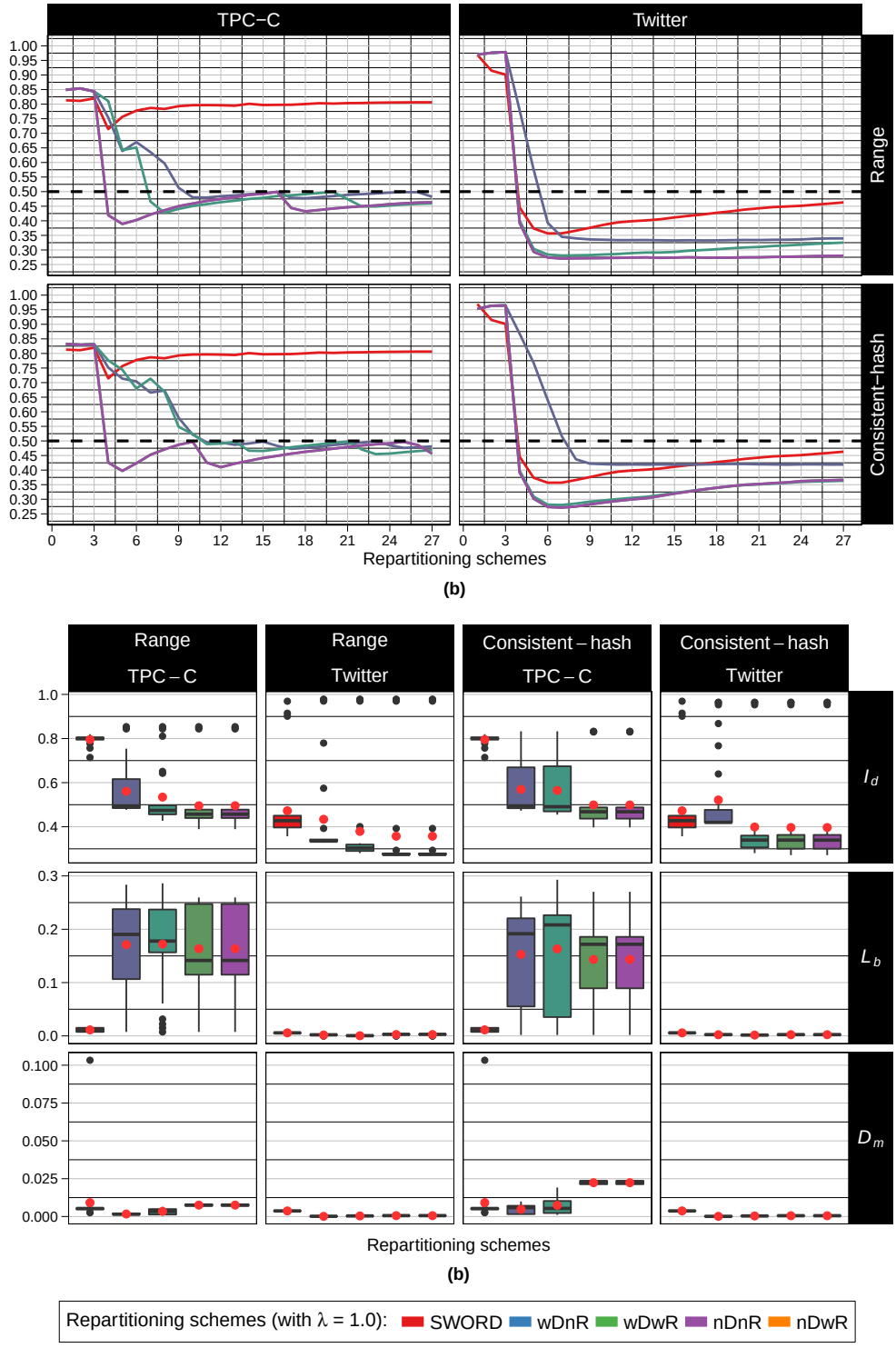


Figure 5.5: Comparison of the proposed greedy heuristic based repartitioning schemes (with $\lambda = 1.0$) and *SWORD* observing I_d, L_b, and D_m under both *TPC-C* and *Twitter* workloads in a *range-* and *consistent-hash partitioned* database cluster, respectively

perform initially for a *TPC-C* workload. All variations of our proposed greedy heuristic based incremental repartitioning scheme perform much better than *SWORD*, a state-of-the-art dynamic incremental repartitioning scheme. Overall, both the *nDnR* and *nDwR* variations aggressively converge I_d below the R_t margin by performing more data migrations compared to the *wDnR* and *wDwR* variations that converge rather slowly over time. However, considering all three repartitioning KPIs, *wDwR* performs better as it keeps the impacts of DTs and number of data migrations low simultaneously, for both of the workload and database cluster types.

5.4.3 Comments on the Experimental Results

To summarise, in situations where we need to immediately reduce I_d, either of the *nDwR* or *nDnR* variations is the preferable choice considering data transfer cost is negligible and one-time large volumes of data migration bursting is tolerated by the system. These two variations will always choose the best migration plan for reducing I_d (for $\lambda = 1.0$), whether we restrict the migration plan generations or nor. Both *wDnR* and *wDwR* variations are most suitable for normal operational conditions as they force I_d to slowly converge towards the R_t threshold with minimum data migrations over time. The *wDnR* variation is best for achieving the highest per data migration net gain for I_d in each repartitioning cycle, and is better suited to systems with limited IOPS resources. On the other hand, if the underlying system can support occasional I/O bursting then the *wDwR* variation is the best option, as it can more aggressively reduce I_d compared to *wDnR* by migrating nearly the same amount of physical data.

From experimental analysis, it is clear that the proposed greedy heuristic based incremental repartitioning scheme can produce near-optimal repartitioning. Furthermore, the convergence of the proposed algorithm can be easily controlled by setting appropriate values for λ and γ. In the graph theoretic schemes, an *MCM* based *TRP* approach is the only one that manages to aggressively reduce I_d below the triggering threshold for both *range* and *consistent-hash* based databases under both a *TPC-C* and *Twitter* workload. However, the L_b KPI seems to suffer in all of those scenarios due to its aggressive nature of pushing more tuples into a single DS using the many-to-one cluster-to-partition mapping. In the greedy heuristic approach, however, it is

possible to prioritise L_b gains by executing migration plans that are optimised for data migrations. Therefore, the L_b KPI performances are much better than the graph *min-cut* based schemes for all settings. The proposed greedy scheme also validates that it is possible to find optimal/near-optimal repartitioning results with an unrestricted unique transaction generation model, while overcoming the shortcomings of the graph based abstractions and increasing network size in real-world scenarios.

5.5 Conclusions

In this chapter, a greedy approach for workload-aware incremental repartitioning of OLTP databases is proposed which produces optimal/near-optimal results using multi-objective KPI requirements. The proposed scheme very clearly outperforms existing exhaustive approaches like *SWORD* by using effective greedy heuristics. By removing the workload network abstraction layer altogether, as introduced in the previous chapter, direct transaction-level optimisations greatly improve the performance of the overall repartitioning process. However, due to the iterative data migrations during each cycle, it is not possible to direct the repartitioning process towards achieving any particular KPI—I_d or L_b. In fact, the repartitioning KPIs I_d and L_b behave semi-orthogonally to each other while physical data migrations take place; therefore, it is often hard to achieve a proper trade-off between them. This has also been clearly observed in the experimental results. For large-scale deployment and real deployments, the search space for a repartitioning problem can still grow large even for the most sophisticated greedy heuristics. Therefore, there is still a need to reduce the repartitioning search space to a level that can only work with the representative transactional snapshots containing the frequently appearing elements in a workload. Finally, an on-demand repartitioning solution can take advantage of this to find an approximate solution to the incremental repartitioning problem that is effective to use in practice. In next chapter, we further discuss these abovementioned issues and present an dynamic repartitioning approach based on transactional stream mining.

Chapter 6

Incremental Repartitioning with Transactional Data Stream Mining

In the previous chapter, a greedy heuristic based approach for incremental database repartitioning has been discussed and evaluated. The scheme has shown effectiveness in achieving the two repartitioning KPIs—the impact of DTs and server-level load-balance with strict control over physical data migrations. However, the number of transactions in any given observation window may create a bottleneck on the performance of such schemes. In this chapter, a transactional *Data Stream Mining (DSM)* based incremental repartitioning scheme is proposed to facilitate effective decision making by analysing only a limited number of highly recurring DTs.

6.1 Introduction

In a dynamic OLTP application, the number of unique transactions is expected to grow with time. By discarding the transactions that were generated before the current workload observation window, the growth rate in the number of unique transactions may be considerably held in check, but it will, nevertheless, be high in the long run. Consequently, scalability of both the graph *min-cut* based repartitioning (Chapter 4) and the greedy optimisation based repartitioning (Chapter 5) schemes can suffer significantly due to incremental computational complexity overload. The running time of a graph *min-cut* algorithm depends on the number of edges as well as vertices in the graph that are directly proportional to the number of unique DTs, along with the MNDTs and the data tuples collectively covered by them, respectively. The computational complexity of the greedy optimisation heuristic depends on these aspects.

An incremental increase in the number of unique transactions may also lead to inferior repartitioning decisions as the proposed schemes do not factor in data popularity,

a phenomenon widely observed in random networks. Not all transactions are alike in popularity, which is driven by either the underlying workload generation model or by external events relevant to the OLTP application. To make sense of all transactions, it is vital to understand how they are being used, both in the workload and by the associated KPIs. To make the right business and operational decisions, it is now common practice to classify data according to a temperature scale of *hot*, *warm*, and *cold*, representing data that are used frequently (dominant), less frequently (commoner), and rarely (outlier), respectively. It has been observed on many types of data studied in the physical and social sciences that the frequency of data is inversely proportional to its rank in the frequency table, referred to as the empirical Zipf law [118]. In general, the number of unique data items classified as hot is a small fraction of the entire set of unique data, while they represent a large fraction of the workload due their high frequency of usage.

This chapter investigates ways of constructing a representative sub-network of representative transactions and of performing transaction-level adaptive repartitioning on this sub-network only. This strategy is expected to keep the computational complexity in check as all the outliers and a large number of commoner transactions are not considered. Concomitantly, the performance of the repartitioning schemes is also expected to improve due to the filtering out of a significant volume of infrequent transactions that have little bearing on the KPIs. Recently, the transactional stream mining of Frequent Itemsets (FIs),[1] i.e., Frequent Tuplesets (FTs), has gained prominence in Big Data applications in order to discover interesting trends, patterns, and exceptions over high-speed data streams. On-line stream mining techniques [165, 166] that incrementally update FIs over a sliding window are the most appropriate tools for achieving the goal of this chapter as their amortised computational complexity [167] is significantly lower. Mapping graph *min-cut* clusters of the sub-network of representative transactions directly to the physical servers also provides an opportunity to develop atomic incremental repartitioning schemes where an individual transaction may trigger repartitioning decisions for the data tuples covered by that transaction based on some server-associativity metric.

The rest of the chapter is organised as follows. The motivations and significance of

[1] The terms *itemset* and *tupleset* are used interchangeably in this thesis.

using transactional DSMs to identify and utilise the frequently occurring transactions for incremental repartitioning is discussed in Section 6.2. An extended transaction classification technique accompanying the graph *min-cut* based repartitioning is presented in Section 6.3, along with experimental evaluation and comparative analysis with the results obtained in Chapter 4. Section 6.4 details the Association Rule Based Hypergraph Clustering (ARHC) [168] based atomic incremental repartitioning scheme which uses the extended transaction classification method from Section 6.3. From experimental results, this adaptive version justifies its effectiveness for practical use. Finally, Section 6.5 concludes the chapter by discussing the potential and limitations of the proposed technique.

6.2 Overview

In this section, the challenges related to incremental repartitioning of large-scale transaction processing systems and possible ways forward are discussed.

6.2.1 The Impacts of Transactional Volume and Dimensions

To demonstrate the adverse effects of increasing transactional volume and dimensions on the graph *min-cut* based clustering process, a three-dimensional empirical analysis is conducted on hypergraph based transactional networks. The *khmetis* graph *min-cut* library is used to perform k-way balanced clustering of the network while varying 1) the number of transactions to 500, 1000, 1500, and 2000; 2) the target number of clusters to 100, 200, 300, and 400; and 3) the transactional dimensions, i.e., varying the number of tuples in a transaction to 5, 10, 15, 20, and 25. In Figure 6.1, the cluster times are presented in bar plots with respect to the varying size of clusters (Figure 6.1(a)) and different number of transactions (Figure 6.1(b)). It can be clearly observed from the figure that clustering time grows at polynomial order with the increase in all three dimensions – transactional dimensions, volume, and target number of clusters. This monotonically increasing time in the clustering process reveals the limitations of using a graph *min-cut* based repartitioning scheme in real-life. Similarly, the computational time complexity of the greedy heuristic based repartitioning approach is

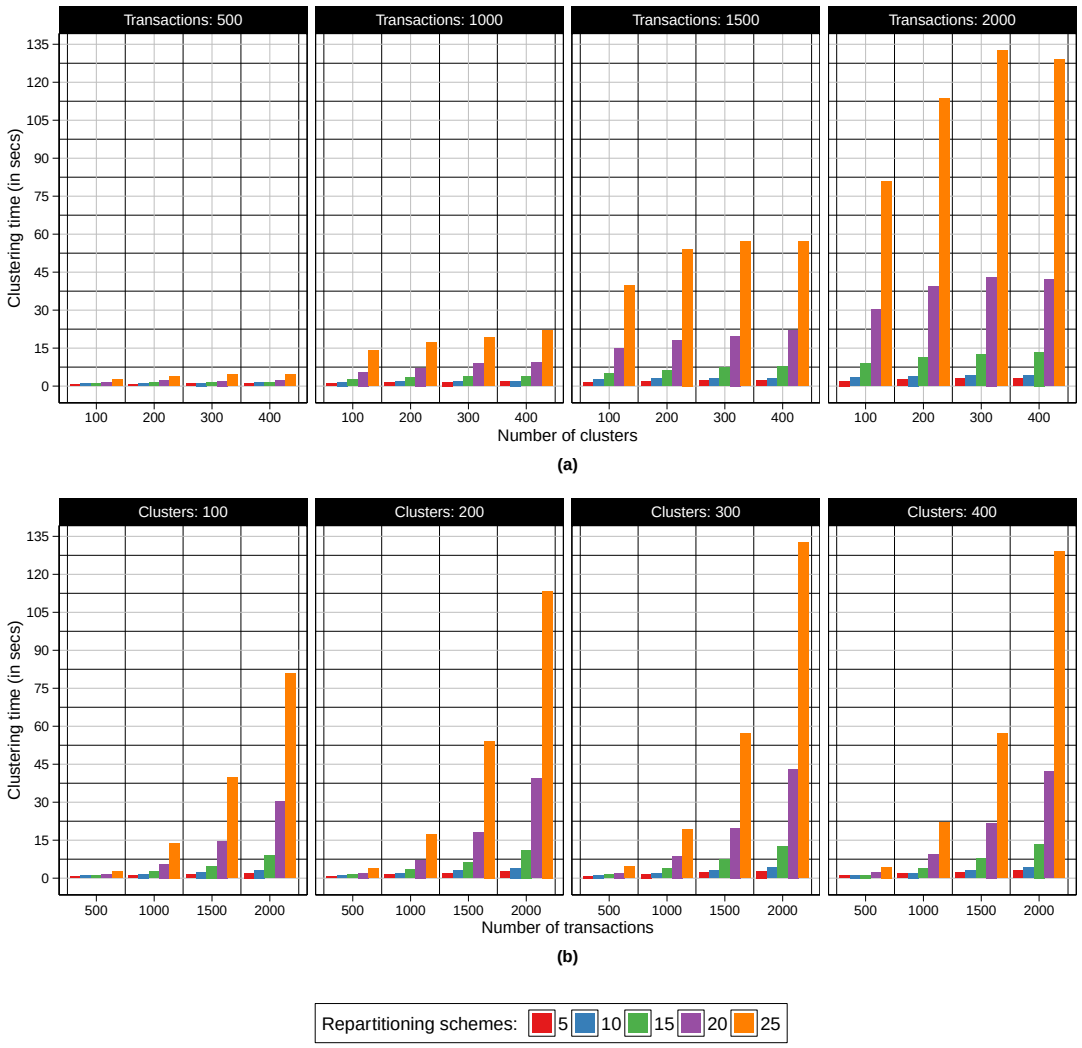


Figure 6.1: The clustering times for transactional hypergraphs with different dimensions relating to a) increasing number of transactions, and b) increasing number of target clusters

also bounded by linearithmic order on the number of DTs as discussed in Section 5.2. Therefore, in practice, it is challenging to obtain either a sub-optimal or near-optimal repartitioning solution when the transactional volume is in the magnitude of millions within an observation window and, at the same time, spanned over several thousands of database partitions and hundreds of physical servers. A natural way to deal with the abovementioned challenge is to determine the *hot* tuples in the database and construct a representative workload network with the transactions containing them. More specifically, by identifying the representative transactions containing the frequent tu-

plesets, the workload size for analysis and processing can be dramatically reduced to make quick repartitioning decisions.

6.2.2 Transactional Stream Mining

Tracking the *hot*, i.e., frequently recurring, tuplesets within the transactional streams from the database logs has been extensively studied [165, 169–171] and used in practice [172, 173]. Over the past years, DSM techniques become extremely popular in understanding the insights of real-time database usage patterns for e-commerce, banking, insurance, retail and manufacturing, and telecommunication applications. Traditionally, transactional logs are first archived and indexed within a data warehouse environment, then data mining techniques are used to understand and retrieve the useful and interesting facts in and about the data. However, transactional queries in modern OLTP systems are generated at an unparalleled high rate that requires on-demand and real-time analytic insights. Real-time predictive analytic services typically use a single pass over the incoming transactional streams to generate the outputs. In cases where the outputs can be useful at the end of an *observational* phase, a *sliding window* is maintained where the predictive outputs are kept updated upon the processing from an individual pass. Depending on the size of these *sliding windows* and types, whether time or transaction sensitive, a trade-off between the quality of the outputs and how quickly predictive decisions are made is determined.

The task of finding the frequently occurring items in data streams is of significance in many application areas. For instance, identifying frequently transmitted packets within an Internet router at any ISP can reveal the users with the highest bandwidth usage or the most popular destinations of its users in the Web. DSM schemes are also used for market-basket analysis, marketing and determining pricing policies, credit card fraud detections, real-time detections, click stream analysis in Web sites, etc. Similarly, DSM techniques can be used to determine the representative set of transactions containing frequently appearing tuplesets in a distributed OLTP database in order to make effective incremental repartitioning decisions without analysing large volumes of transactional logs.

6.2.2.1 Frequent Tupleset Mining in a Transactional Database

Let $\bar{\delta}_i \in \wp(D)$, where $\wp$ denotes the power set, be any arbitrary tupleset and $\mathcal{T}_{\bar{\delta}_i}$ be the set of transactions containing $\bar{\delta}_i$. The *support* and *relative support* of $\bar{\delta}_i$ can be defined, respectively, as follows:

$$sup(\bar{\delta}_i, T) = |\mathcal{T}_{\bar{\delta}_i}| \tag{6.1}$$

$$\widetilde{sup}(\bar{\delta}_i, T) = \frac{|\mathcal{T}_{\bar{\delta}_i}|}{|T|} \tag{6.2}$$

Given a *Minimum Support Threshold (MST)* ∂, $\bar{\delta}_i$ is called *frequent* i.e., a Frequent Tupleset (FT) in T, if and only if, $\widetilde{sup}(\bar{\delta}_i, T) \geq \partial$.

In the literature, *Apriori* [174] is one of the oldest algorithms to find frequent tuplesets from a static set of transactions in an iterative way. By performing a level-wise search using tuplesets of k length, the algorithm finds the set of candidate patterns with the length of $k+1$. This iterative process continues until all the frequent tuplesets with the longest possible length are found and no further frequent tuplesets exist. However, this static process is not suited to iteratively scanning very large candidate sets generating long frequent tuplesets. Later, Han *et al.* [175] utilised *frequent pattern trees (FP-tree)* for storing frequent patterns in a compressed way, and a *frequent pattern growth (FP-growth)* technique to find the complete set of frequent patterns by concatenating the frequent 1-itemsets iteratively. Despite its superiority over the original *Apriori* approach, where there are high transactional volumes and dimensions, the iterative search spaces can exponentially grow over time along with the number of FIs.

To further optimise this process, *frequent closed tuplesets (FCTs)* are introduced. A tupleset $\bar{\delta}_i$ is said to be *closed* if there exist no frequent superset $\bar{\delta}_j$ such that

$$\nexists\, \bar{\delta}_j \supset \bar{\delta}_i \mid \widetilde{sup}(\bar{\delta}_j) = \widetilde{sup}(\bar{\delta}_i) \tag{6.3}$$

therefore, the set of *FCTs* only contains unique elements and is typically much smaller than the set of FIs. Most importantly, the set of FTs can be calculated from the *FCIs* as a tupleset will be frequent if, and only if, it is a subset of any *FCI*. This approach allows the *FCI* mining algorithms to save computing time and space and allow for non-

duplicate outputs. The particular problem of finding *FCIs* from a set of transactions in a static database has been extensively studied in the literature [176–179].

6.2.2.2 Frequent Tupleset Mining in Transactional Streams

The problem of finding frequent tuplesets in the transactional streams is similar to the problem discussed above, except that the search space is not a static dataset anymore, but a *sliding window* over the stream that slides forward over time. The window can be either *time-sensitive* – it contains transactions arriving within the last t time units, or *transaction-sensitive* – it contains the last n transactions independent of the transaction generation rate $\mathcal{R}$. In this thesis, the *transaction-sensitive* approach is adopted for mining *FCIs* in the stream, as it is more practical to define a transaction batch size for a fixed $\mathcal{R}$ to be analysed in practice. Furthermore, it helps in comparing the results in a simulation environment, rather than having varying a number of transactions over fixed size time-sensitive sliding windows. Let W be the transaction-sensitive sliding window containing the set of fixed number of transactions, and $|W|$ be the window size. Therefore, (6.2) can be adopted for W as

$$\widetilde{sup}(\bar{\delta}_i, W) = \frac{|\mathcal{T}_{\bar{\delta}_i}|}{|W|}. \tag{6.4}$$

The process of *FCI* mining in the transactional streams can be either *exact* or *approximate*. Exact mining needs to preserve all the *FCIs* in W in order to track the infrequent tuplesets that might become frequent over time. On the other hand, the approximate process can contain false positives and/or false negatives as it does not store *FCIs* from the past. In most real-life scenarios, approximate outcomes are acceptable as the exact solutions can become impractical for large Ws and high values of $\mathcal{R}$. In this thesis, an approximate *FCI*/*FCT* mining approach is taken based on the above analogy. Interested readers can read further about other methodologies for an exact *FCI* mining processes using a time-sensitive sliding window in [165, 166, 180–184].

To further optimise the approximate mining technique, Cheng *et al.* [185] propose *semi-FCIs* where the *MST* of an *FCI* is increased over time as it constantly appears within successive Ws. This *relaxed-MST* or *maximum support error* is used to prevent

the dropping of a frequent tupleset from the solution when it becomes infrequent over few slides of W. Based on this novel notion of *semi-FCI*, the authors proposed *IncMine*, an approximate *FCI* mining algorithm, that uses an update per transaction batch policy over a time-sensitive sliding window model in order to incrementally update the set of *semi-FCI*s over high-speed data streams. A high-level description of the *IncMine* algorithm will be presented here for the readers to understand the iterative *semi-FCI* update process which is later incorporated with the proposed incremental repartitioning algorithms.

In its simplistic form, the objective of *IncMine* is to find the closed *FCIs* over W. Let $\mathcal{F}$ be the set of *FCIs* determined over W in a particular instance. As a new batch of transactions arrive, $\mathcal{F}$ needs to be updated to incorporate the new transactions and compensate the effects of the old transactions from the last batch update process. It first searches the *semi-FCIs* from the latest transactions, and then updates L with the latest findings. Consider, a tupleset $\bar{\delta}_i$ is marked as an *FCI* during the last batch update with $\widetilde{sup}(\bar{\delta}_i, T) \geq \partial$, but becomes infrequent in the latest W. It is possible that $\bar{\delta}_i$ may become frequent again, i.e., by obtaining relative support of at least $\partial|T|$ in the next slide. However, it is not guaranteed, thus, $\bar{\delta}_i$ can be silently dropped from $\mathcal{F}$.

In such situations, a *relaxation factor*, $r \in [0,1]$ can be used to check whether the tuplesets in W have at least ∂r support to be stored in L as an *FCI*. *IncMine* extends this idea by dynamically increasing the value of r for a $\bar{\delta}_i$ if it was an *FCI* in the previous observations. As *IncMine* uses an update per transaction batch policy, let B be the fixed size transaction batch where n number of Bs constitute W i.e., $W = nB$. Now, for $k \in \{1,...,n\}$, let $minsup(k)$ be the incremental minimum *MST* function for $\bar{\tau}_i$ as

$$minsup(k) = \lceil \partial r_k \rceil; \tag{6.5}$$

where

$$r_k = \frac{1-r}{n-1}(k-1) + r \tag{6.6}$$

which gradually increases the value of r for each succeeding B. Note that, $r_1 = r$ and $r_n = 1$. Any $\bar{\tau}_i$ is kept in the *semi-FCI* list if, and only if, its approximate support over the most recent Bs is no less than $minsup(k)$ i.e., $\widetilde{sup}(\bar{\tau}_i) \geq minsup(k)$.

Table 6.1: The key DSM configuration parameters used in *OLTPDBSim*

Sliding window type	Transaction-sensitive
Sliding window size, $\|W\|$	$3\|\mathcal{W}\|$
Transaction batch size, $\|B\|$	$\|\mathcal{W}\|$
Frequent tupleset mining	*CHARM* [186] implementation using Java *BitSet*
Transactional stream mining	*IncMine* [185]
Minimum support threshold (MST), ∂	0.1
Initial relaxation rate, r	0.5
Maximum tupleset length	Disabled – will perform full stream mining

Let us now formally define *representative* transactions (RTs) that are used in the proposed repartitioning schemes presented in this chapter. A transaction τ is referred to as an RT only if there exists an *FCI* $\bar{\delta}_i$ such that $\bar{\delta}_i \in \tau$. To distinguish RTs from the set of all transactions T, we use the notation $\bar{\tau}$ to denote an RT and the set of all RTs is denoted by $\mathcal{T}$, $\mathcal{T} = \{\bar{\tau}_1, ..., \bar{\tau}_{|\mathcal{T}|}\} \in T$.

6.2.3 Transactional Stream Mining in *OLTPDBSim*

To incorporate DSM into *OLTPDBSim*, an efficient implementation of the *IncMine* algorithm is used from [187] which has been developed on top of the existing *MOA (Massive On-line Analysis)* [172] platform. For mining the tuplesets in successive transactional streams the particular *IncMine* package in *MOA* uses an efficient implementation of the *CHARM* algorithm [186] using *BitSet*[2] for finding the FIs. The stream collector implementation uses *EvictingQueue*[3] to construct a circular FIFO for storing the transactional streams. This FIFO implements the sliding window W which contains a fixed number of transactional batches, Bs. Most of the *IncMine* implementations are fork lifted from *MOA* [172] under its open source license and properly modified to integrate into the *Incremental Repartitioning* module of the *OLTPDBSim* simulator as shown in Figure 3.3. When transactional stream collection is enabled in the simulation configuration file, the stream collector FIFO continuously collects transactional logs.

[2] Java *BitSet* is a data structure of dynamically resizing array of bits.

[3] EvictingQueue is imported into *OLTPDBSim* from Google's Guava library, which is a fixed-size queue that automatically removes elements from the head when attempting to add elements to a full queue.

During each trigger of the repartitioning process, the entire FIFO is passed to the *IncMine* process as a sliding window, and the algorithm then processes and updates the transactions per batch as defined in the configuration. The DSM output consisting of the *semi-FCIs* are then used to find the set of representative transactions $\mathcal{T}$. Finally, a representative workload network is constructed from $\mathcal{T}$ which is much smaller than any of the original graph based networks previously used in this thesis. Table 6.1 refers to the key DSM parameters for *IncMine* implementation used in *OLTPDBsim* in order to utilise transactional stream mining to facilitate the incremental repartitioning decision. As an example, according to Table 6.1, the values of $|W|$ and $|B|$ will be set to 10800 and 3600, respectively, for $\langle \mathcal{R}, \mathcal{W}, p \rangle = \langle 1, 3600, 0.15 \rangle$. With these settings, the possible values for r can be 0.5, 0.75, and 1, thus the set of $minsup(k)$ values to 0.05, 0.075, and 1 for the *IncMine* process in order to dynamically adjust the MST values for transactional steam mining.

6.3 Transactional Classifications using Stream Mining

In this section, transactional stream mining is used to extend the proactive transaction classification scheme discussed in Section 4.4.

6.3.1 Extended Proactive Transaction Classifications

The notion of *moveable* NDT (MNDT) has been introduced in the proactive transaction classification technique to create a workload network consisting of DTs and NDTs that have at least one tuple in a DT. This special sub-network is then used in the graph *min-cut* based incremental repartitioning process to prevent those NDTs from turning into DTs over time. By continually mining the *semi-FCIs* from the transactional sliding windows, DTs and NDTs in the classification tree can be further classified by introducing the notions of distributed *semi-FCTs* (DFCTs) and non-distributed *semi-FCTs* (NDFCTs). These selective sets of DTs and MNDTs are then used to construct a representative workload network, which is much smaller in size compared to a network containing every DT from $\mathcal{W}$. The obvious drawback to this approach is the slow convergence of I_d towards the repartitioning triggering threshold R_t during each

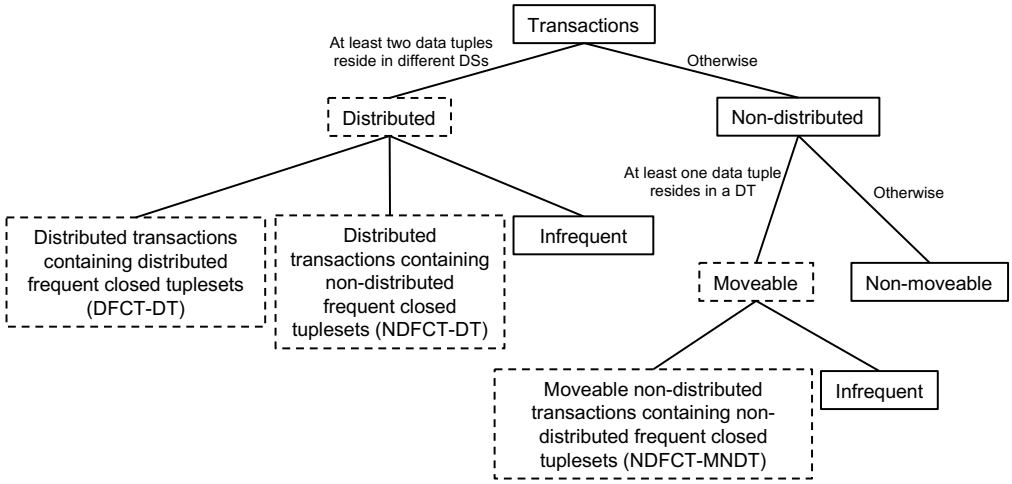


Figure 6.2: The proactive transaction classification process enhanced by identifying the frequent closed tuplesets in the transactional workload

repartitioning cycle, which also influences the server-level data distributions. Yet, in an approximate solution, the benefits from faster workload network processing with fewer transactions can compensate for the abovementioned shortcomings over time.

Based on the above heuristic, an extended proactive classification tree is presented in Figure 6.2. Transactions are classified as DTs and NDTs at the top-level. NDTs are further classified into *moveable* and *non-moveable* groups. The *moveable* type contains at least a single tuple which is also present in a DT, while the *non-moveable* type represents pure NDTs. By utilising the DSM technique, both DTs and moveable NDTs are further classified on the basis of containing DFCIs and NDFCIs, i.e., *semi-FCIs*. All other DTs and NDTs are classified as *infrequent*. From the above classifications, three distinctive proactive transaction classification techniques can be derived as

(1) *No Transaction Classification (NTC)*—includes all DTs only

(2) *Base Transaction Classification (BTC)*—includes all DTs and MNDTs (as proposed in the original proactive transaction classification technique described in Section 4.4)

(3) *Partial Representative Transaction Classification (PRTC)*—only includes DTs and MNDTs that contain distributed *semi-FCTs* and *moveable* but non-distributed *semi-FCTs* (i.e., DFCT-DTs and NDFCT-MNDTs)

(4) *Full Representative Transaction Classification (FRTC)*—only includes DTs and MNDTs that contain any of the *semi-FCTs* (i.e., DFCT-DTs, NDFCT-DTs, and NDFCT-MNDTs).

6.3.2 Experimental Evaluation

In this section, the proposed extended proactive transaction classification techniques are compared using the same graph theoretic incremental repartitioning schemes proposed in Chapter 4. Experimentations are performed following a *reactive* and *threshold*-based (TRP) approach. The incremental repartitioning process starts once the system I_d increases for a pre-defined value of repartitioning triggering threshold R_t. Transactional logs observed within the preceding $\mathcal{W}$ are then used to create a workload network as either *GR*, *CGR*, *HR*, or *CHR*. This workload graph is then clustered and mapped back to the database partitions based on the proposed cluster-to-partition mapping techniques detailed in Section 4.6. We decided to use the *MCM* mapping technique as it exhibited superior incremental repartitioning performance in the experimental evolutions of Section 4.7.

By enabling transactional DSMs in *OLTPDBSim*, the proposed *PRTC* and *FRTC* techniques are implemented; later their effectivenesses is compared with the *NTC* and *BTC* approaches for graph theoretic incremental repartitioning. We will refer to these four different repartitioning approaches as *NTC-MCM*, *BTC-MCM*, *PRTC-MCM*, and *RTC-MCM*. Both *TPC-C* and *Twitter* workloads are simulated using a restricted transaction generation model for *range-* and *consistent-hash partitioned* database clusters each consisting of four physical DSs. The necessary simulation parameters are used as listed in Tables 4.2 and 6.1.

6.3.2.1 Analysis of Experimental Results for *TPC-C* and *Twitter* Workloads

In this section, the experimental results for both *range-* and *consistent-hash partitioned* database clusters under *TPC-C* and *Twitter* workloads are presented and discussed.

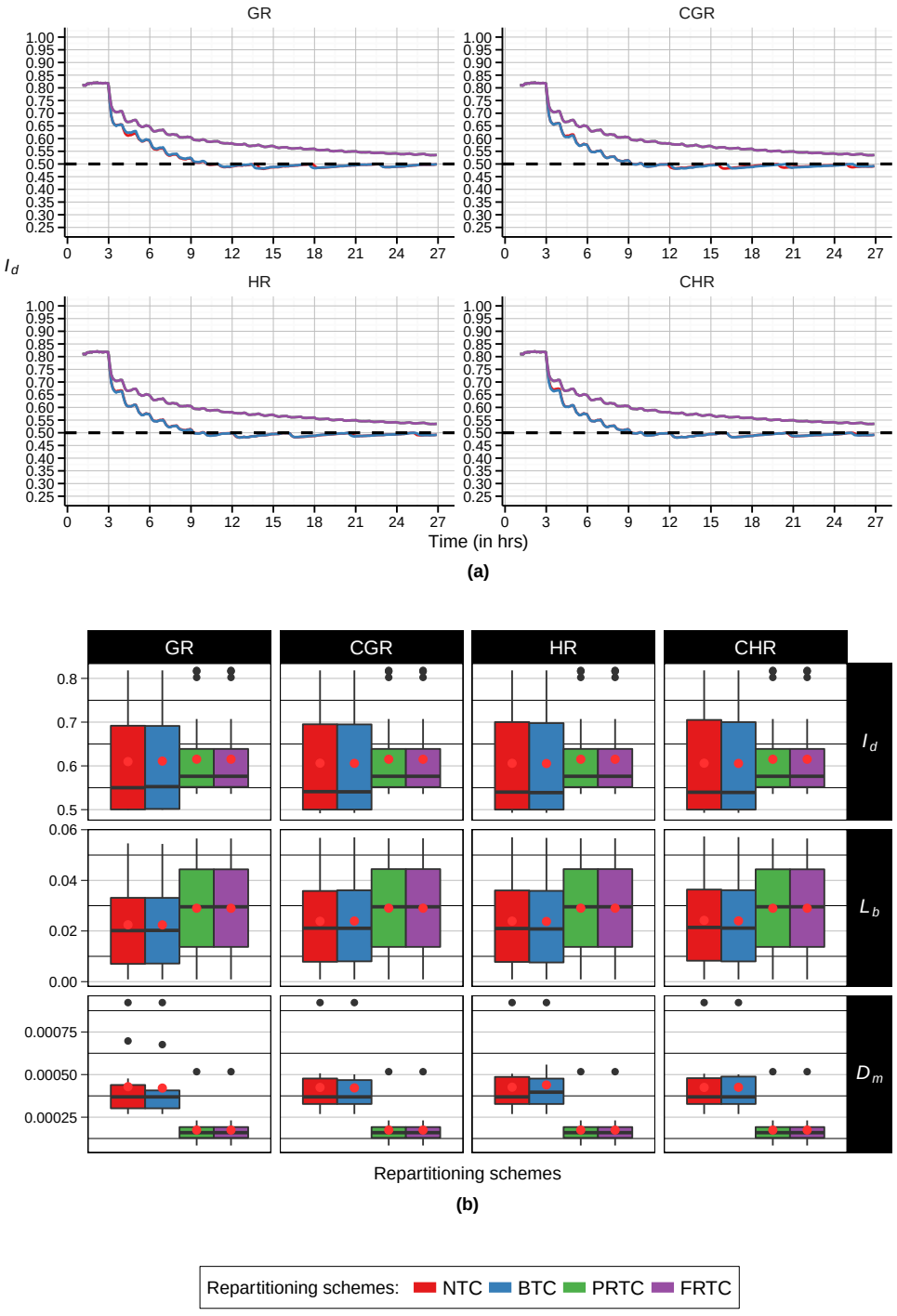


Figure 6.3: Comparison of graph *min-cut* based incremental repartitioning schemes with different transaction classifications in a *range-partitioned* database under a *TPC-C* workload

6.3.2.1.1 Range-Partitioned Database

Figure 6.3 presents the simulation results comparing the *NTC-MCM*, *BTC-MCM*, *PRTC-MCM*, and *FRTC-MCM* techniques in a *range-partitioned* database driving *TPC-C* workloads. Figure 6.3(a) shows the performance of I_d for each of the workload network representations of *GR*, *CGR*, *HR*, and *CHR*. The results reveal that there are hardly any observational differences in the repartitioning KPIs between using *NTC-MCM* and *BTC-MCM*, or between *PRTC-MCM* and *FRTC-MCM* techniques. The frequent tupleset mining technique slowly reduces I_d towards the threshold, as was expected. This is due to the approximate solutions obtained from each of the incremental repartitioning cycles. Similarly, Figure 6.3(b) presents the results for all three repartitioning KPIs—I_d, L_b, and D_m—using box plots, and again the results are hardly distinguishable between *NTC-MCM* and *BTC-MCM*, or between *PRTC-MCM* and *FRTC-MCM*. One of the noticeable distinctions in the L_b performances for *PRTC-MCM* and *FRTC-MCM* are that these are slightly worse compared to *NTC-MCM* and *BTC-MCM*. Figure 6.4 presents the results for a *Twitter* workload in the same *range-partitioned* database. The results are almost identical in terms of distinguishing between using *NTC-MCM* and *BTC-MCM*, as well as between *PRTC-MCM* and *FRTC-MCM*. However, as shown in Figure 6.4(b) both L_b and D_m performances for *PRTC-MCM* and *FRTC-MCM* are much better compared to *NTC-MCM* and *BTC-MCM*, while I_d performances are almost identical in all cases.

6.3.2.1.2 Consistent-Hash Partitioned Database

Figures 6.5 and 6.6 present the experimental results comparing *NTC-MCM*, *BTC-MCM*, *PRTC-MCM*, and *FRTC-MCM* in a *consistent-hash partitioned* database driving *TPC-C* and *Twitter* workloads, respectively. According to Figure 6.5(a), I_d performances do not vary much between *NTC-MCM* and *BTC-MCM*, or between *PRTC-MCM* and *FRTC-MCM*, for *GR* and *CGR* networks. For the *HR* network, the distinctions between different transaction classifications is very clear. The *NTC-MCM* technique makes rapid progress in reducing I_d, while *BTC-MCM*, *PRTC-MCM*, and *FRTC-MCM* are all making slow progress, respectively, dropping I_d below the threshold. Similar trends are also noticeable for *CHR* networks. From Figure 6.5(b), the L_b and D_m performances of the *PRTC-MCM* and *FRTC-MCM* techniques are better than *NTC-MCM* and *BTC-MCM* as they were also making slow progress in reducing I_d.

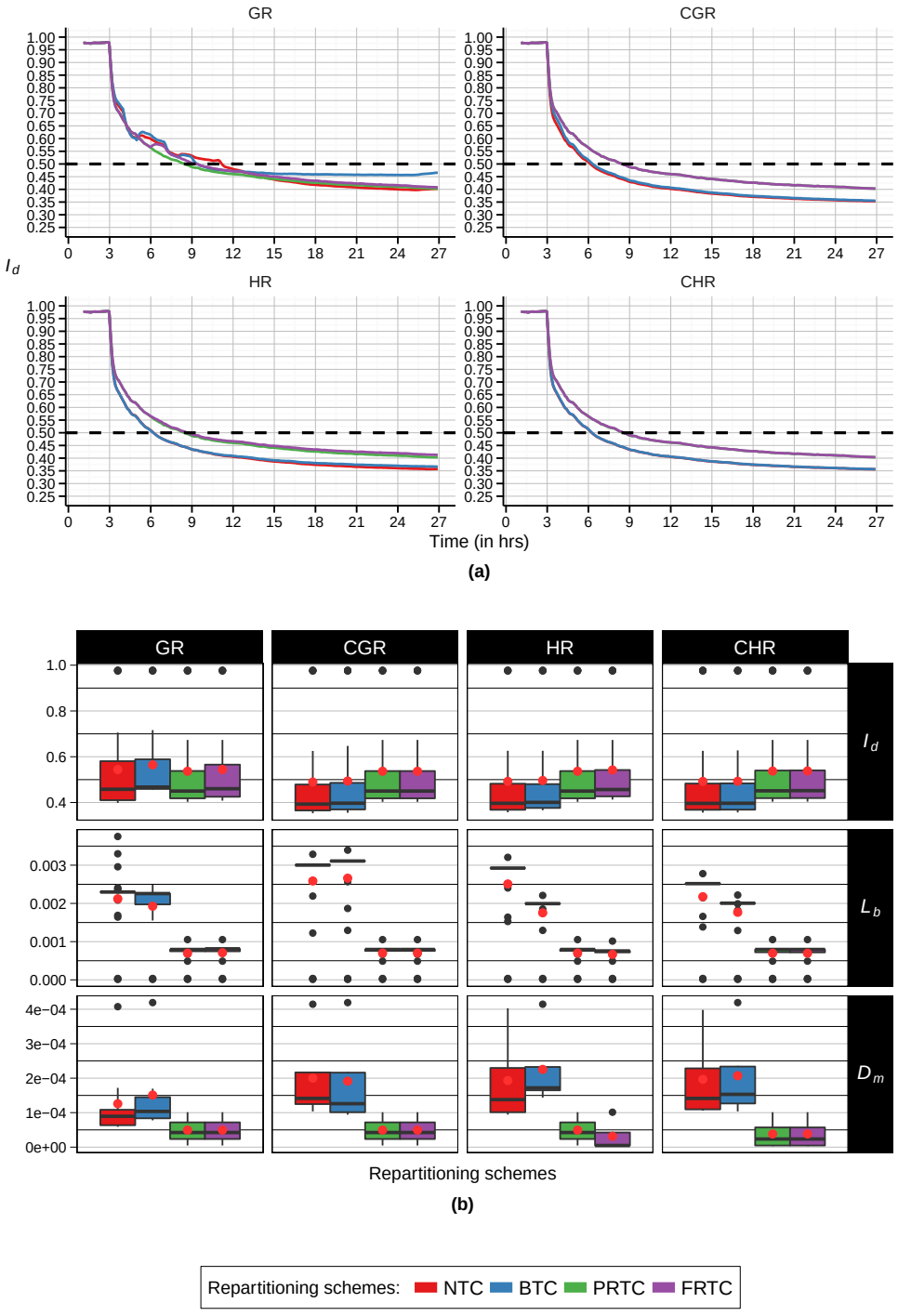


Figure 6.4: Comparison of graph *min-cut* based incremental repartitioning schemes with different transaction classifications in a *range-partitioned* database under a *Twitter* workload

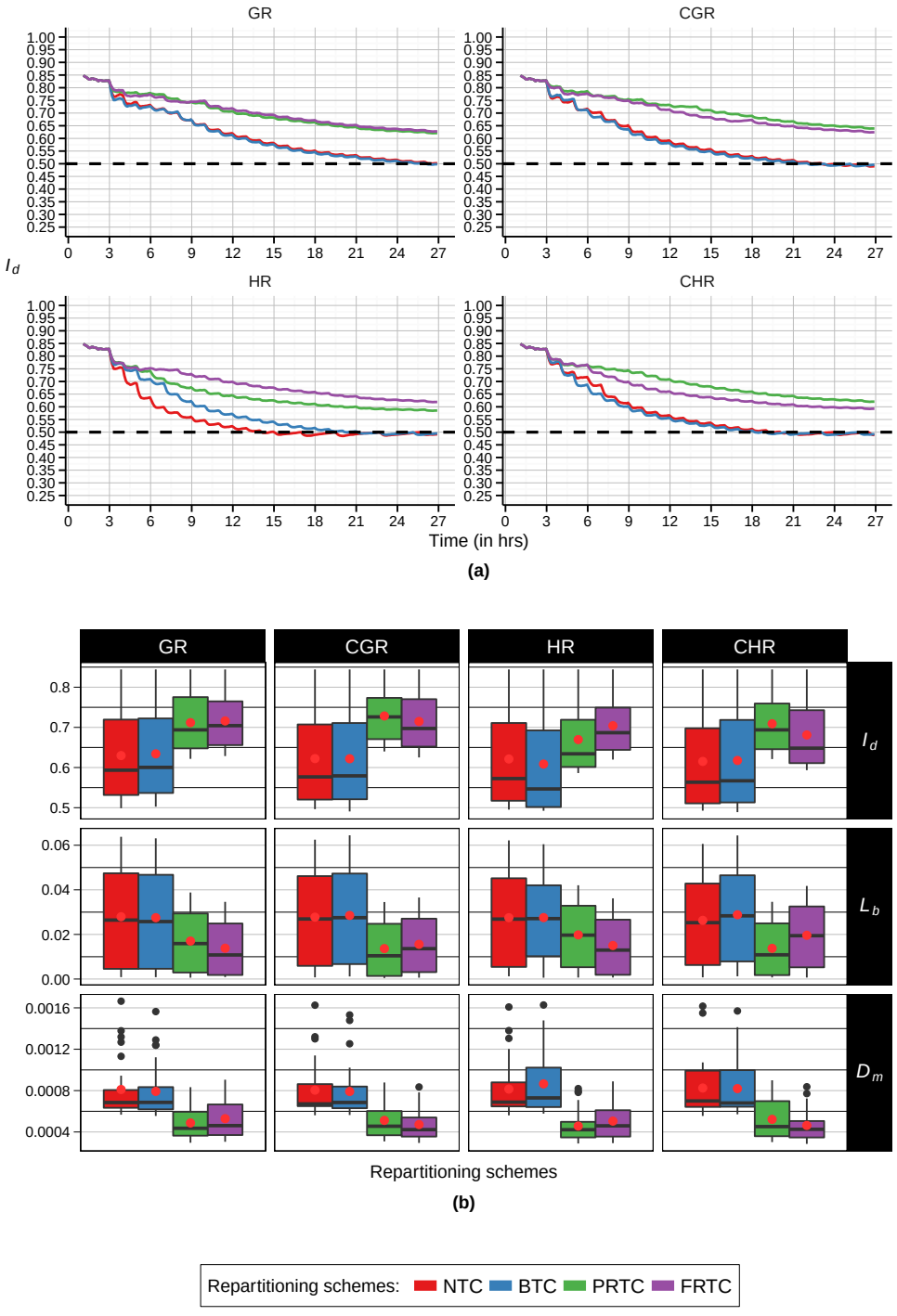


Figure 6.5: Comparison of graph *min-cut* based incremental repartitioning schemes with different transaction classifications in a *consistent-hash partitioned* database under a *TPC-C* workload

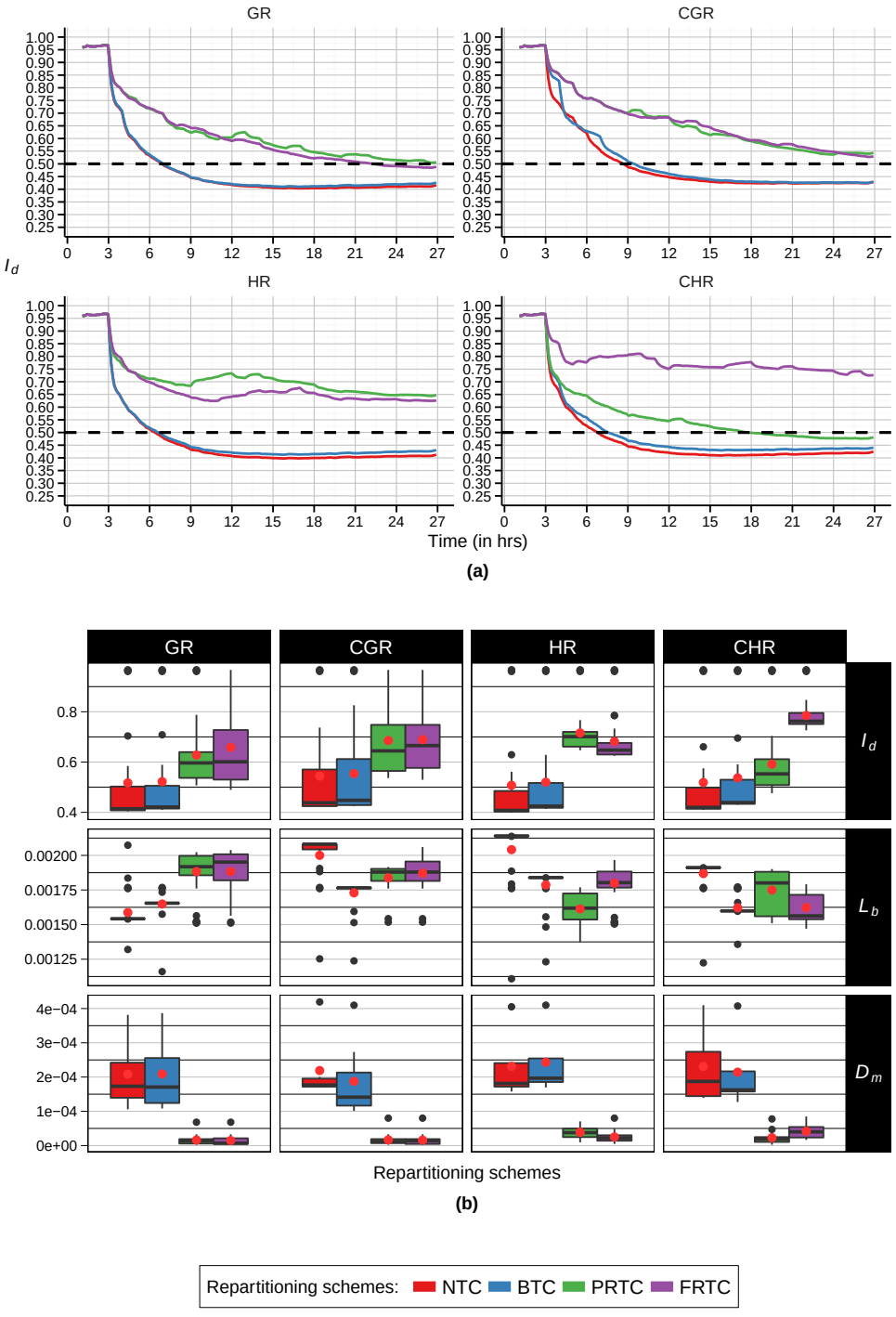


Figure 6.6: Comparison of graph *min-cut* based incremental repartitioning schemes with different transaction classifications in a *consistent-hash partitioned* database under a *Twitter* workload

While using *Twitter* workload, as shown in Figure 6.6(a), only *GR* and *CGR* networks show a declining trend in I_d over time. For the *HR* network, both the *PRTC-MCM* and *FRTC-MCM* techniques fail to reduce I_d below the threshold over a simulation period of 24 hours. In contrast, the *PRTC-MCM* technique for *CHR* representation managed to reduce I_d gradually but *FTC-MCM* failed to reduce I_d over time. As shown in Figure 6.6(b), the L_b and D_m performances for the *PRTC-MCM* and *FRTC-MCM* techniques are very similar with few physical data migrations, which explains why both these techniques took longer in reducing I_d.

Overall, the *BTC-MCM* technique performs better than other techniques; however, it certainly requires processing more transactions compared both the *PRTC-MCM* and *FRTC-MCM* techniques, as expected. As a concluding remark, it is obvious that the DSM based extended transaction classification techniques require less transactional processing while compensating repartitioning KPIs over time. Therefore, such graph theoretic approaches do not have the quality to be applied in real-life deployments. In the next section, a transactional stream mining technique is exploited to enhance the greedy heuristic based incremental repartitioning scheme for applying in practice.

6.4 Atomic Incremental Repartitioning

The experimental results from the previous section reveal the inability of the transaction classification process alone to fully utilise the benefits of DSM. The primary observation is that each incremental repartitioning cycle tends to perform better when the input workload network contains all the DTs and MNDTs, in contrast to the network which only contains a selective set of transactions that contains only the frequent tuplesets. An obvious drawback here is that, as the workload network size grows, the ability to run the incremental repartitioning cycles on-demand and more frequently is lost. Therefore, there is a clear need to ensure a compact representative workload network as an input to the incremental repartitioning problem. This might not output an optimal repartitioning result, but the related KPIs will eventually catch up over time. Most importantly, this enables us to run incremental repartitioning cycles more frequently and in an on-demand basis.

In the previous section, the frequent tuplesets are directly used to identify the representative set of DTs and NDTs for constructing the input graph or hypergraph network. This is a fine-grained approach to utilise the associative knowledge of clustered tuplesets in incremental repartitioning. However, the same approach can be used to discover high-level knowledge by constructing a representative hypergraph network from the frequently appearing tuplesets found during the transactional tupleset mining process. By performing *min-cut* clustering on this compact hypergraph, it is possible to group the set of all the frequently occurring tuples together in transactions. Furthermore, it distributes the transactional data access load within the entire database cluster. This clustering of tuplesets can be particularly useful in classifying incoming transactions based on their data access patterns.

A similar technique has been reported as successful in the literature for performing *Association Rule Hypergraph Clustering (ARHC)* [168, 188, 189]. In *ARHC*, the knowledge (i.e., pattern) represented by these clusters is used to classify the actual transactions into different groups to understand the database usage patterns by the users. For example, clusters of items from the shopping carts in an on-line retail Web application can help in classifying the group of customers who have young children. This same idea can be adopted to repartition incoming transactions based on their maximum association to a particular cluster containing a set of frequent tuples.

In light of the repartitioning problem, the set of frequent tuplesets, i.e., *semi-FCTs*, can be mined using the transactional stream mining technique and referred to as *association rules* in this context. These *semi-FCTs* can be further used to create a hypergraph network, where an individual *semi-FCT* can denote a hyperedge and its contained tuples represent the overlaid vertices. The frequency counts of an individual *semi-FCT* denote the weight of the hyperedge. Vertex weight in a hyperedge is represented by its data size (in volume), which is considered to be the same for all tuples in the database for simplicity. This hypergraph representation is equivalent to an *Association Rule Hypergraph (ARH)* network mentioned above. Then, a graph-cut library like *HMETIS* [85, 90] is used to perform *ARHC* to produce balanced clusters and redistribute them back to the DSs.

As the representative *ARH* network will be much smaller than a graph or hypergraph network containing all the DTs and MNDTs from an observing $\mathcal{W}$, it is, there-

fore, better to map the clusters to the physical servers instead of the logical database partitions (as we did in Chapter 4). The clusters can be mapped to the DSs using the same cluster-to-partition mapping techniques—*RM*, *MCM*, and *MSM*. This process can be run every time the system wide I_d increases above R_t, and the results can be stored to perform transaction-level atomic repartitioning for each of the transactions that arrives.

Following an *ARHC* process, an on-demand atomic repartitioning decision is made for each incoming transaction. Now, to select a data migration plan for an individual transaction, its *associativity* is measured against all the *ARH* clusters. Based on the highest *associativity* value, the residing DS of the particular *ARH* cluster is chosen as the destination server for many-to-one data migrations. If a transaction does not show any *association* with any of the *ARH* clusters through its *associativity* measure, it is simply marked as *processed* and no repartitioning takes place. Furthermore, in order to avoid the *ping-pong effect*, the same heuristic is borrowed from Chapter 5, that works over a particular observation window. This is to ensure that an atomic repartitioning does not affect its preceding incident transactions that have been already marked processed by repartitioning scheme, observed over the same time window.

Note that, in this proposed scheme, there are no explicit controls to keep any particular repartitioning KPIs–I_d, L_b, and D_m—in check. However, grouping transactions into DSs, based on their frequent access patterns (i.e., association with a *ARH* cluster), can successfully reduce the overall impact of DTs in the system (as will be observed in the experimental results). Furthermore, this atomic approach can approximately distribute the server-level load based on transactional access patterns. Thus, server-level load-balance is also maintained as a repartitioning by product. In addition, unlike the repartitioning schemes presented in Chapters 4 and 5, which require decisions and the performing of all necessary data migrations at a single point in time, the atomic repartitioning can distribute the computational cycles, disk operations, and network I/Os over the entire observation window. Therefore, the overall cost of the repartitioning process can be amortised over time. This is By far the most scalable approach as well, compared to the graph theoretic and greedy heuristic repartitioning schemes considering the volume of transactional logs to be stored and analysed. Most importantly, this novel approach will allow the system administrator to activate and de-activate

on-demand incremental repartitioning during any observation window.

6.4.1 Proposed Atomic Repartitioning Scheme

In this section, we will discuss how the abovementioned idea can be implemented as an on-demand atomic repartitioning scheme using the *ARHC* technique. Let $\Lambda = \{\bar{\delta}_1, ..., \bar{\delta}_{|\Lambda|}\}$ be the set of all *semi-FCTs* mined over $\mathcal{W}_i'$ to create an *ARH* $\widetilde{\mathcal{H}}$. To perform *ARHC* on $\widetilde{\mathcal{H}}$, where a k-way balanced *min-cut* process is used with a load balancing factor β_k based on (4.1) where $k = |S|$, let $\widetilde{V} = \{V_1, ..., V_k\}$ be the set of *ARH* clusters obtained from $\widetilde{\mathcal{H}}$, where each V_i contains the set of vertices representing the frequently accessed data tuples observed over $\mathcal{W}_i'$. Finally, the *ARH* clusters are distributed among k DSs following either of the *RM*, *MCM*, and *MSM* techniques.

To perform atomic repartitioning, each incoming τ_i will be classified under *FRTC* based on the extended proactive transaction classification tree shown in Section 6.3.1. Let us define the associativity function for any τ_i as

$$A_c(\tau_i, V_i) = \frac{w(V_j)}{\sum_{l=1}^{k} w(V_l)} \frac{|\tau \cap V_j|}{\sum_{l=1}^{k} |V_l|}. \tag{6.7}$$

An individual *associativity* score is taken for all the *ARH* clusters in order to find the residing DS S_i of V_i which has the highest *associativity* value for τ_i. Once the destination DS has been decided, then the entire tupleset of τ_i is migrated to S_i by performing many-to-one data migrations. If τ_i is a DT, then after this migration it will be converted to an NDT; otherwise, it will remain a DT. By applying the greedy heuristic described in Section 5.2.3.2, atomic repartitioning is skipped if the intended data migrations are likely to affect any of the preceding incident transactions that have been already processed within this $\mathcal{W}_i'$ to avoid the *ping-pong effect*. This atomic repartitioning process continues until the systemwide measure of I_d goes below the repartitioning threshold. Once I_d rises above this threshold, the *ARHC* process takes effect and utilises the ongoing transactional streams to find the frequently appearing *FCTs* in the current workload observation window.

6.4.2 Experimental Evaluation

This section reports on simulation based experimentations conducted to compare the transactional DSM based *PRTC-MCM* and *FRTC-MCM* schemes with the proposed *ARHC* based atomic incremental repartitioning scheme. We use similar experimental setups to those in Section 6.3.2 with the simulation parameters listed in Tables 4.2 and 6.1 for *OLTPDBSim* except that this is an unrestricted transaction generation model. In the experimentations, the incremental repartitioning process begins once the systemwide I_d increases over R_t at any point during the database lifecycle. The set of *semi-FCTs* over $\mathcal{W}$ are then used from the continuous transactional log stream mining process, and to construct the *ARH* network. This compact *ARH* subgraph is then partitioned into four clusters (as four DSs are used in the experiment) using the *HMETIS* [90] hypergraph clustering library. These clusters are then placed into the underlying physical DSs following the mapping techniques proposed in Section 4.6.

Incoming transactions are classified using *FRTC* based on the extended classification tree presented in Figure 6.2. Individually selected DTs are then evaluated on their A_c scores measured against the *ARH* clusters (based on (6.7)). Once a destination DS is decided based on the highest A_c score for a DT, the *many-to-one* data migration process then converts it to an NDT in an atomic step.

For ease of comparison, we denote the *ARHC* based atomic incremental repartitioning schemes as *ARHC-RM*, *ARHC-MCM*, and *ARHC-MSM* depending on the *ARH* cluster-to-server mapping technique that is used. In the following sections, the KPIs of the abovementioned atomic incremental repartitioning techniques are then compared with the previously evaluated *PRTC-MCM* and *FRTC-MCM* schemes outlined in Section 6.3.2.

6.4.2.1 Analysis of Experimental Results for *TPC-C* and *Twitter* Workloads

In this section, the experimental results for a *range-* and *consistent-hash partitioned* database cluster under the *TPC-C* and *Twitter* workloads are presented and discussed.

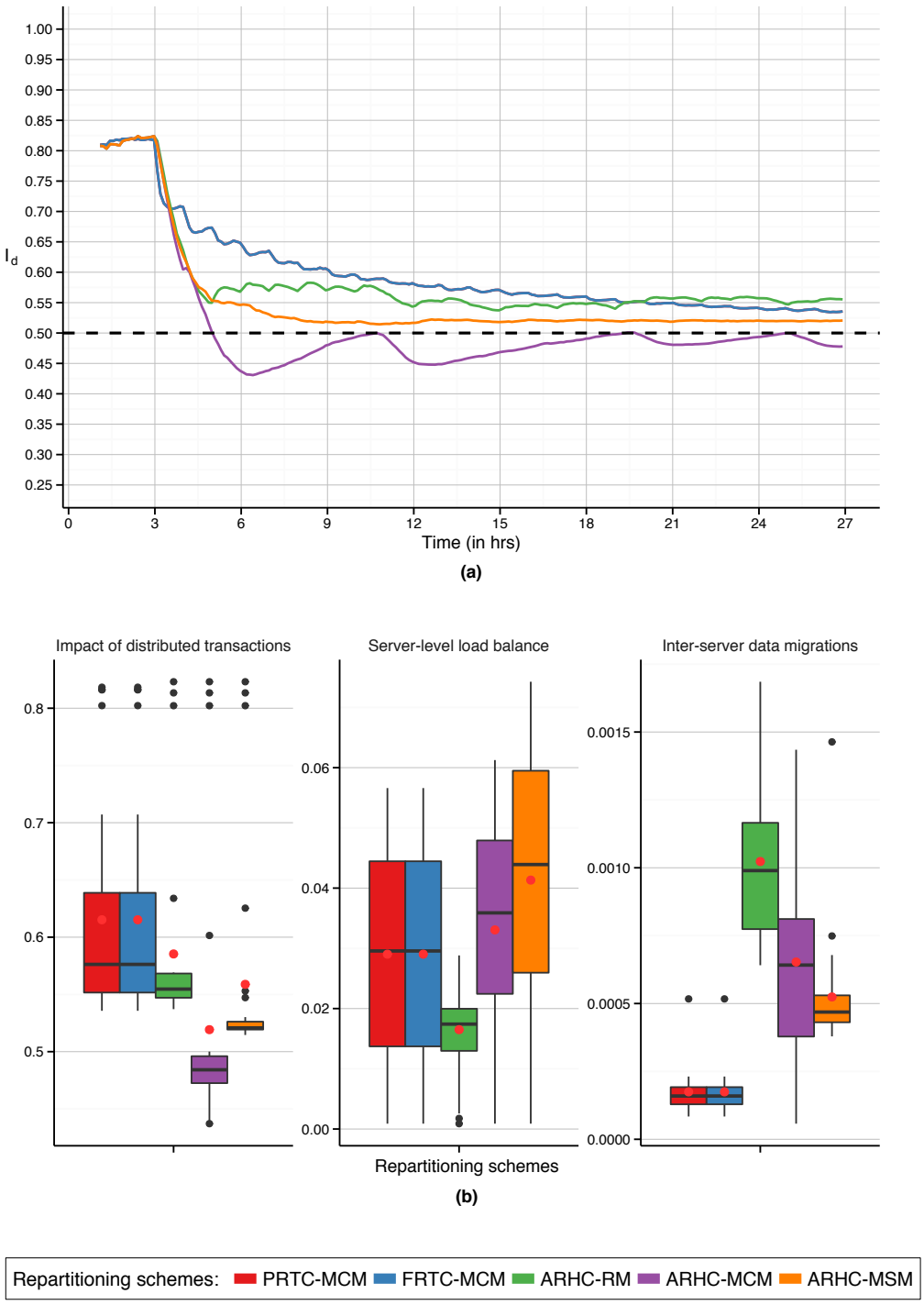


Figure 6.7: Comparison of different transactional data stream mining based repartitioning schemes in a *range-partitioned* database for observing I_d, L_b, and D_m under a *TPC-C* workload

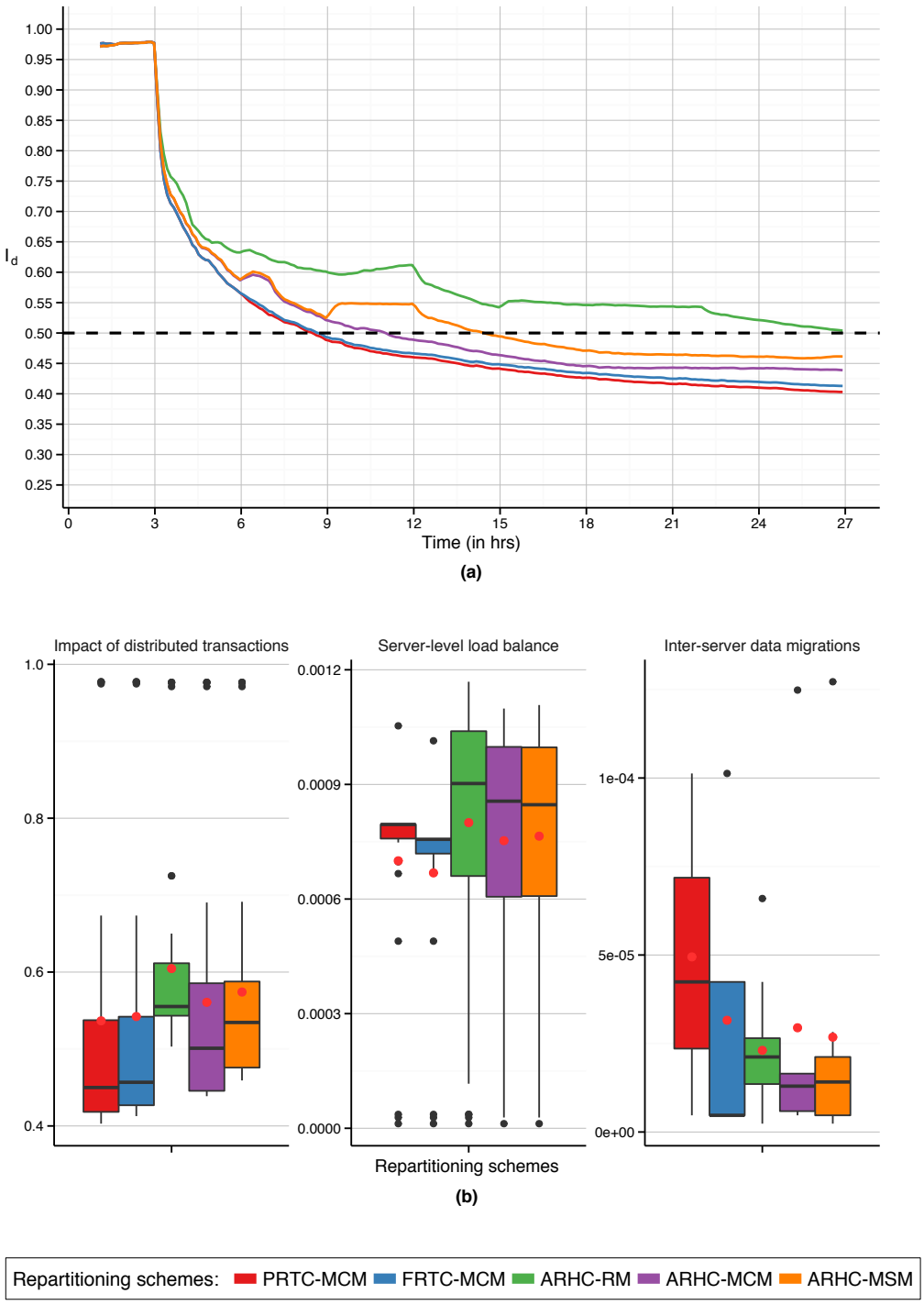


Figure 6.8: Comparison of different transactional data stream mining based repartitioning schemes in a *range-partitioned* database for observing I_d, L_b, and D_m under a *Twitter* workload

6.4.2.1.1 Range-Partitioned Database

Figure 6.7 presents the experimental results of the *ARHC* based atomic incremental repartitioning schemes—*ARHC-RM*, *ARHC-MCM*, and *ARHC-MSM*—in a *range-partitioned* database under the *TPC-C* workload. Figure 6.7(a) compares the I_d performances of the proposed schemes. Based on the results, *ARHC-MCM* performs aggressively compared to others and quickly converges to bring down I_d under R_t within a few repartitioning cycles. The other two *MCM* based schemes—*PRTC-MCM* and *FRTC-MCM*—show very slow convergence over the period of 24 h and are almost inseparable in the line plots. The *ARHC-RM* and *ARHC-MSM* schemes do not converge well, rather they maintained a steady level of I_d just above the R_t margin. Figure 6.7(b) presents the overall I_d, L_b, and D_m performances using box plots. In all cases, *ARHC-MCM* shows promising results to maintain adequate levels of all three KPIs. The *ARHC-MCM* scheme acts more aggressively compared to others and performs more data migrations during each of the atomic repartitioning operations, but the overall L_b performance manages to spread out over time due to occasional repartitioning triggering. On the other hand, the *ARHC-RM* scheme maintains L_b by performing more data migrations while the *ARCH-MSM* scheme performs fewer selective data migrations and does the same task in a better way.

Experimental evaluations under a *Twitter* workload are presented in Figure 6.8. Figure 6.8(a) presents the I_d performance while Figure 6.8(b) shows the overall I_d, L_b, and D_m characteristics of the compared schemes. Due to the specific implementation of our *Twitter* workload, the *Follower* and *Follows* relationships stabilise once the frequently appearing *Followers* are placed into the same physical server, and then rarely migrate from there. Therefore, in the case of *MCM* based schemes—*PRTC-MCM*, *FRTC-MCM*, and *ARHC-MCM*—once I_d values drop below the preset $R_t = 0.5$ within a few incremental repartitioning cycles, no further repartitioning cycles are needed. For *ARHC-RM*, as the representative data tuples migrate randomly into the physical servers over consecutive repartitioning cycles, the transition of I_d towards the R_t threshold is much slower. Finally, in the case of *ARHC-MSM* mapping, this abovementioned process is much slower due to the added restrictions on inter-server data migration policy. This explains the L_b and D_m characteristics in the results shown in Figure 6.8(b).

6.4.2.1.2 Consistent-Hash Partitioned Database

Figure 6.9 presents experimental results for the proposed *ARHC* based atomic incremental repartitioning schemes compared with *PRTC-MCM* and *FRTC-MCM* within a *consistent-hash partitioned* database driven under a *TPC-C* workload. Figure 6.9(a) compares the I_d performances of the evaluated schemes. Among these techniques, only *ARHC-MCM* manages to reduce I_d below R_t more aggressively than others. While *ARHC-RM* and *ARHC-MSM* maintain a steady level of I_d, both *PRTC-MCM* and *FRTC-MCM* schemes show slow convergence towards R_t. As expected from an *MCM* based technique, *ARHC-MCM* triggers aggressive data migrations bursts to reduce I_d whenever it is necessary, therefore it converges very quickly compared to others. Figure 6.9(b) shows the overall I_d, L_b, and D_m performances of the experimental results using box plots. It is interesting to notice that, although *ARHC-MCM* performs more data migrations than any of the other schemes compared, it can still maintain a better L_b due to its occasional triggering of the repartitioning cycles.

Finally, Figure 6.10 presents the experimental results for the proposed schemes in a *consistent-hash partitioned* database under a *Twitter* workload. As can be observed from Figure 6.10(a), all the *ARHC* based schemes show good convergence in bringing I_d down under the R_t threshold over time. Unlike the performances of the *PRTC-MCM* and *FRTC-MCM* schemes in a *range-partitioned* database under a *Twitter* workload observed in Figure 6.8(a), I_d performances show less convergence in this instance. This is due to how the *Twitter* database is distributed in a *range-* and *consistent-hash partitioned* database. By performing the graph theoretic *PRTC-MCM* and *FRTC-MCM* repartitioning schemes in a *range-partitioned* database, the most frequently occurring tuples can be grouped into the same physical DS very quickly with the *MCM* mapping technique. However, for a *consistent-hash partitioned* database, it is not possible to utilise this DSM knowledge when the representative workload networks from the *PRTC* and *FRTC* classifications are clustered.

Figure 6.10(b) presents the overall I_d, L_b, and D_m performance of the schemes compared in box plots. Simulation results show similar behaviours to those observed in other cases under a *Twitter* workload. Due to the unique characteristics of the *Twitter*

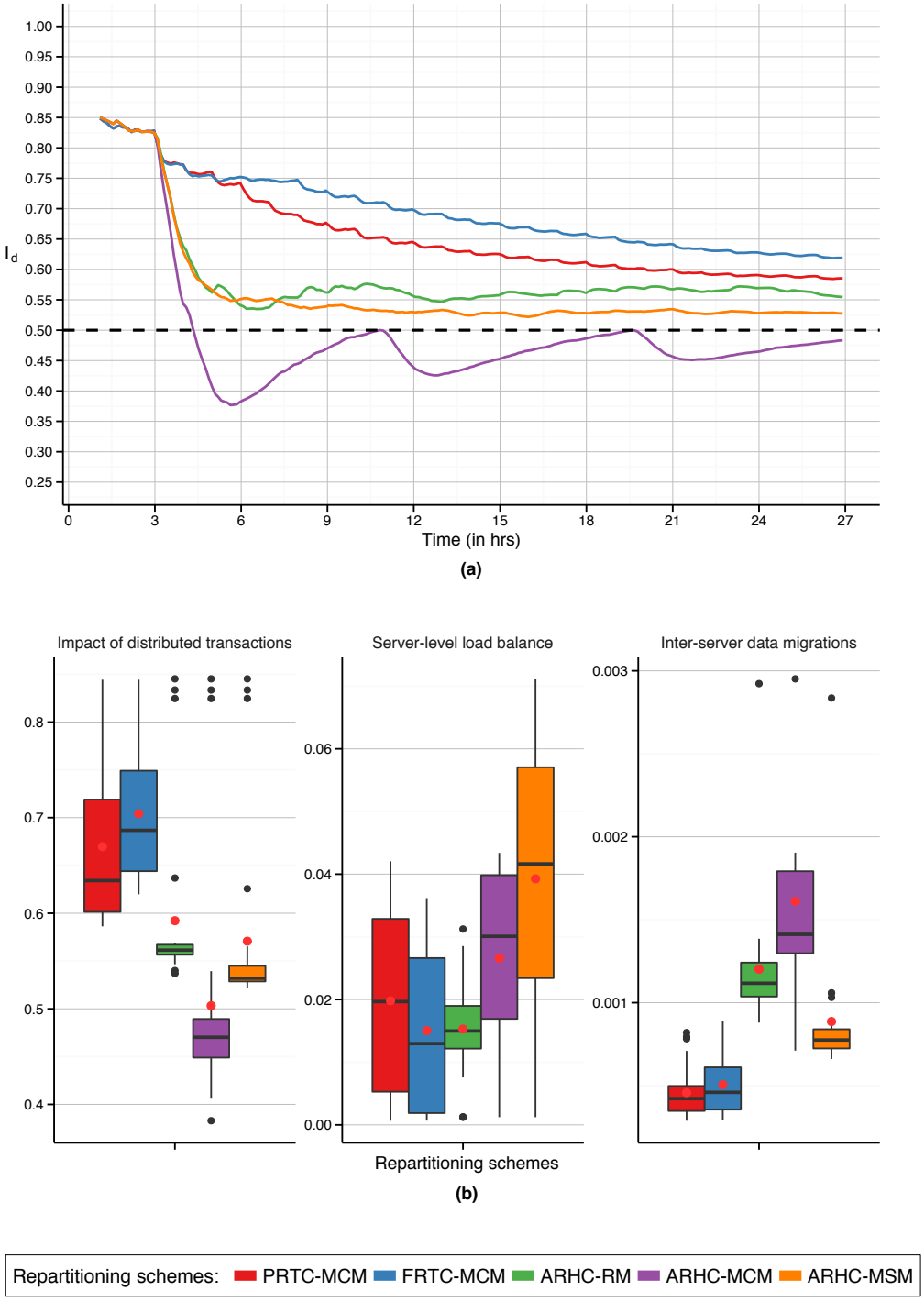


Figure 6.9: Comparison of different transactional data stream mining based repartitioning schemes in a *consistent-hash partitioned* database for observing I_d, L_b, and D_m under a *TPC-C* workload

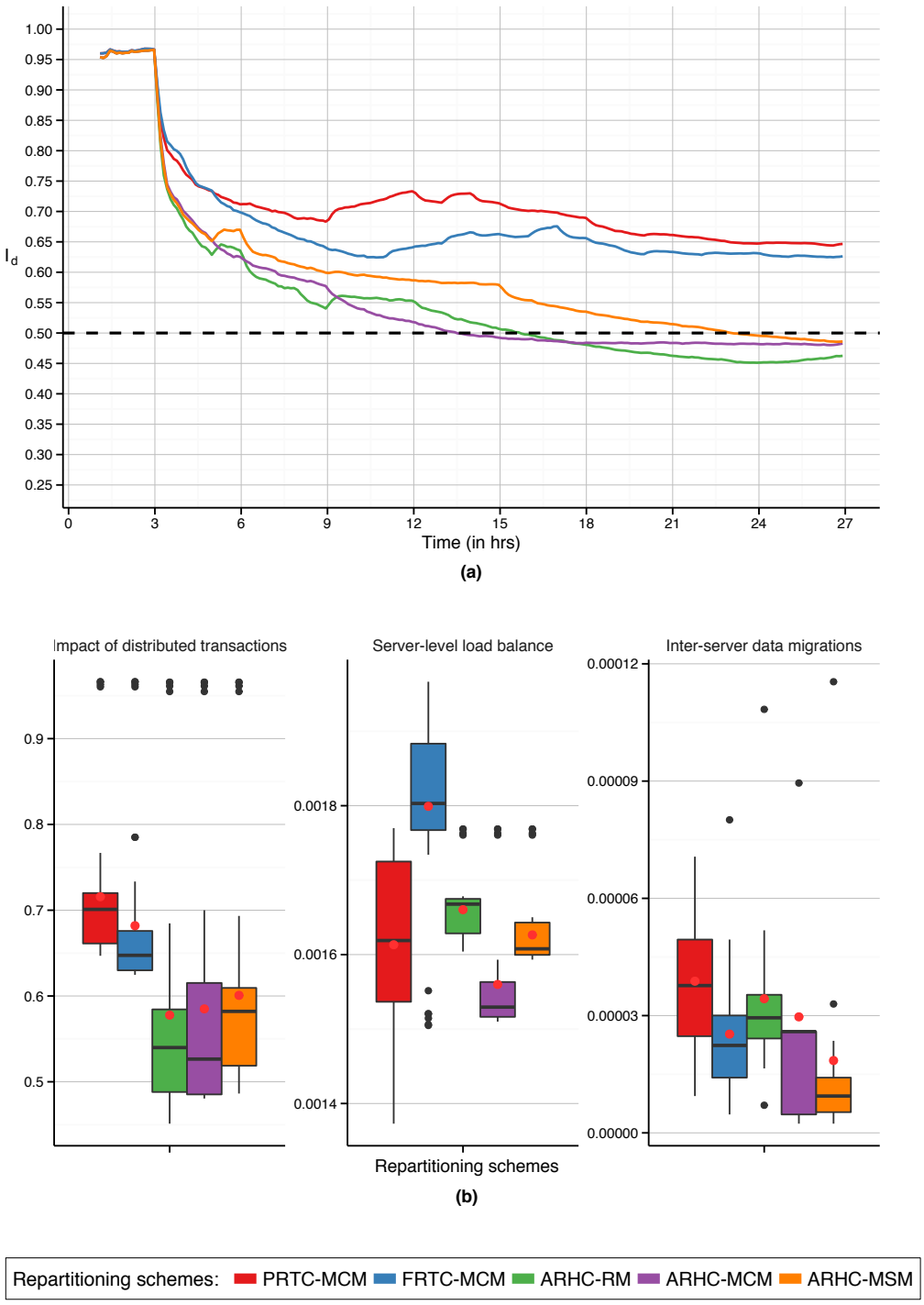


Figure 6.10: Comparison of different transactional data stream mining based repartitioning schemes in a *consistent-hash partitioned* database for observing I_d, L_b, and D_m under a *Twitter* workload

workload, once the frequently occurring tuplesets are identified, and repositioned in a designated DS, then all the other closely *associated* tuplesets just follow to the same destination during its atomic repartitioning. Whhen these representative tuplesets are randomly (*RM*) or semi-randomly (*MSM*) distributed within the DSs, then the *ARHC-RM* and *ARHC-MSM* schemes progress slowly over time compared to *ARHC-MCM*.

6.4.2.2 Comments on the Experimental Results

From the analysis of experimental results presented in Section 6.3.2, it is obvious that the extended transactional classification based graph theoretic incremental repartitioning schemes are not as capable of converging I_d as fast as can be expected in many instances of OLTP applications. In practice, however it is necessary to perform incremental repartitioning in such a way that converges quickly to reduce the impacts of DTs. In the first approach, the fine-grained knowledge of transactional classifications is not completely utilised when used within a higher-level abstraction of graph networks. Therefore, in the latter approach, representative tuplesets from transactional logs are used to create lower-level associativity knowledge that can be directly utilised by the transaction-level atomic repartitioning process. This is supported by the experimental results presented in Section 6.4.2. The core repartitioning decision is made during each *ARHC* run, which is less time consuming due to the compact size of the representative workload network, thanks to the effective integration of the transactional stream mining technique in *OLTPDBSim*. Therefore, the overall incremental repartitioning time is amortised over any observed $\mathcal{W}$.

Furthermore, the experimental results presented in Chapter 5 suggest that the greedy heuristic based repartitioning schemes that converge quickly are better suited for systems that can support burst IOPS operations, as an individual repartitioning cycle needs to perform large volumes of physical data transfer over the network. However, the proposed atomic incremental repartitioning, *ARHC-MCM*, in particular, does not require such IO bursting requirements from a system as the data migration operations are spread out over an entire observation window $\mathcal{W}$.

6.5 Conclusions

In this chapter, it has been shown conclusively how transactional data stream mining can play a significant role in aiding incremental database repartitioning. By continuously mining the representative tuplesets i.e., frequently appeared transactional tuplesets, from successive $\mathcal{W}$s, a high level workload network can be constructed during an individual repartitioning cycle. These networks can then be used to construct an adaptive transaction classification tree for performing graph *min-cut* based repartitioning using a selective set of DTs and MNDTs. However, experimental evaluations show that, by using the representative set of transactions, the graph *min-cut* scheme with *PRTC* and *FRTC* classifications slowly converge towards the pre-defined R_t based on the approximate repartitioning solutions obtained from each cycle. Next, the transactional DSM technique is further utilised to generate compact workload networks based on *ARH* and later used to perform *ARHC*, in order to obtain high-level knowledge on how to effectively partition incoming transactions so as to reduce the adverse impacts of DTs. Experimental results show that the clustering of *ARHs* is particularly useful in classifying transactions according to their *associativity* with a set of tuplesets that frequently appear together in the workload. This process eventually helps to convert DTs into NDTs within a given $\mathcal{W}$ by performing on-demand atomic repartitioning. The latter scheme has shown superior performance improvements over the graph *min-cut* and greedy heuristic based schemes by providing an approximate repartitioning solution. The ability of the *ARHC* based repartitioning scheme to perform on-demand transaction-level atomic repartitioning with minimum computational footprint and maximum effectiveness in reducing the adverse impacts of DTs, surely proves it to be a suitable candidate for production deployments. In next chapter, we conclude the thesis by summarising and discussing the research contributions.

Chapter 7

Conclusions and Future Directions

This chapter summarises the research contributions on workload-aware data management in shared-nothing distributed OLTP databases presented in the thesis and discusses the key findings. It also mentions the research challenges open in this particular domain and points to future research directions from the thesis.

7.1 Conclusions and Discussion

Modern OLTP applications operating at the Web scale require high scalable transaction processing to deal with large volumes of end-users' requests simultaneously. In response to this challenge, shared-nothing distributed OLTP databases are used to provide horizontal data partitioning in order to increase database write scalability. However, this scalable approach can still suffer when a large proportion of the user queries need to access database records from multiple physical DSs to process the transactions, i.e., DTs. Processing DTs in such a way can create significant scalability bottlenecks, impacting the the entire database cluster. Furthermore, initial database partitioning based on design phase decisions will not be capable of adapting to OLTP workload variations in the future. It is, therefore, necessary to properly understand the impacts of DTs within an OLTP system and express them as measurable KPIs. By intelligently facilitating workload-aware incremental database repartitioning it is possible to reshuffle only the workload-related data tuples by performing on-the-fly data migrations so as to reduce the adverse impacts of DTs.

In this context, we have investigated the abovementioned incremental repartitioning problem and proposed novel ways to deal with it. From a top level view, the objectives of this research have been 1) to develop repartitioning KPIs with appropriate physical interpretations, 2) to use high-level workload network abstractions in order to

study and extend existing graph theoretic approaches to performing incremental repartitioning which provide sub-optimal results, 3) to avoid constructing network abstraction layers and use greedy heuristics to find near-optimal incremental repartitioning decisions, and 4) to utilise data stream mining so as to identify the representative set of transactions from the workload and employ transactional associativity to perform on-demand atomic repartitioning of the incoming DTs.

We first discuss the application and database level inherent challenges of a distributed OLTP system supporting multi-tier Web applications, and later examine existing literature that deals with the repartitioning of such DDBMSs, in Chapter 2. There has been a significant revolution in database research and developments over the last decade in the post-Internet era. To capture that, a detailed discussion has been presented for the readers to understand the challenges related to modern OLTP applications, scalable transaction processing, and the underlying shared-nothing distributed database domain and specific use-cases. Based on this study, the existing literature has been classified and notable contributions are presented highlighting their strengths and shortcomings. Finally, a conceptual framework of the workload-aware incremental repartitioning process is presented to identify the gaps in the existing literature and how they fit into this framework.

In the process of developing the conceptual framework, Chapter 3 has presented the overall system architecture necessary for an incremental repartitioning scheme, the workload properties, and important repartitioning KPIs. Two fundamental challenges in adopting a workload-aware repartitioning scheme in an OLTP database are on-the-fly live data migrations and data lookup. We have solved these problems by proposing a novel distributed data lookup technique that can take a maximum of two lookups (one lookup in most cases via caching the location information) to locate a data tuple within the entire database cluster. Furthermore, we have adopted the concepts of *roaming* from mobile wireless networks and a partition-level data lookup mechanism to support on-the-fly live data migrations without compromising data consistency and write scalability. This provides the base foundation to utilise an RDBMS like a high scalable NoSQL data store while adopting both *range* and *consistent-hash* based initial data partitioning. Chapter 3 has also formally defined the incremental repartitioning problem and presents a novel transaction generation model to aid simulation based

experimental evaluations. The proposed model has been shown to be successful in restricting a fixed proportion of unique transaction generations which is necessary so as to evaluate graph theoretic repartitioning schemes by avoiding any undue external effects. Finally, three unique KPIs have been presented and formulated with their physical interpretations to measure the systemwide impacts of DTs (I_d), server-level data distribution load-balance (L_b), and inter-server physical data migrations (D_m).

In the first stage of developing an incremental repartitioning solutions, graph theoretic schemes have been presented in Chapter 4, that use high-level network abstractions by adopting graph, hypergraph, or their compressed forms to hide underlying transactional complexities. We have proposed a novel proactive transactional classification technique that includes *moveable* NDTs (MNDTs) in the workload network clustering process to prevent them becoming DTs over successive repartitioning cycles. In the graph theoretic approach, the impacts of DTs are reduced by the balanced graph *min-cut* clustering process and there are no explicit ways to control the amount of physical data migrations over time. We have proposed two unique cluster-to-partition mapping techniques—*MCM* and *MSM*—to aid this challenge, that perform many-to-one minimum data migrations and one-to-one data migrations without affecting load-balance, respectively. Furthermore, both *proactive* and *reactive* incremental repartitioning schemes have been evaluated which show the efficacy of the proposed techniques for different workload network representations. Overall, the compressed hypergraph based *MCM* technique *CHR-MCM* has performed better in adapting to different workload and database types. However, due to the use of higher-level abstractions in such graph theoretic approaches, there has been no certainty to confirm whether the repartitioning KPIs—L_b and D_m—are near- or sub-optimal or not.

One of the primary challenges in the graph theoretic approach is that the graph sizes can grow extensively large and this can increase the graph *min-cut* based clustering time in polynomial order. In Chapter 5, a greedy heuristic based incremental repartitioning scheme has been developed which works directly at the transaction-level without using any network abstractions to find an optimal or near-optimal repartitioning outcome. In this approach, for individual transactions, all possible many-to-one migration plans are computed and the best plan is chosen based on the maximum weighted net improvement of I_d and L_b gains. The effective use of the proposed greedy

heuristic has ensured that by selectively transforming a DT into NDT it is possible to prevent *ping-pong* effects within its incident transactions. Furthermore, the proposed scheme has shown its ability to converge on-demand by controlling the possible number of migration plan generations and performing data migration optimised migration plan executions. Based on experimental results, the greedy incremental repartitioning scheme has been shown efficient in adopting a wide variety of OLTP application cases and system-level resource availability for occasional IOPS bursting or fixed IOPS usage at regular intervals. However, the computational complexities of such a greedy approach can still grow with the increase in transactional volumes and dimensions, as well as the database cluster size.

To deal with the abovementioned challenges, a transactional stream mining technique has been used to convincingly find representative transactions to perform incremental repartitioning in Chapter 6. Inherently, this adoption has solved another important challenge of performing on-demand incremental repartitioning. First, an extended transaction classification tree has been proposed to incorporate the knowledge of transactional classifications into graph theoretic incremental repartitioning. However, due to the use of workload network abstractions it is not fully possible to exploit this knowledge to perform repartitioning effectively in a scalable way in practice. Later, an *Association Rule Hypergraph Clustering* based atomic incremental repartitioning scheme has been proposed which can fully leverage the knowledge derived from the clustering of the compact representative network of transactional tuplesets. An individual transaction can atomically transform from DT to NDT by simply associating itself to a particular representative cluster stored in a particular DS. This innovative approach has been found effective through experimental evaluations to be less computationally in finding the approximate repartitioning solution for aggressively reducing the adverse impacts of DTs. As the computational complexities and physical resource usage to find this approximate solution are amortised over an entire workload observation window, the proposed scheme, *ARCH-MCM* scheme in particular, to be successful for practical adoption.

Finally, based on the above discussion, the following key observations and their implications can be directly derived from this thesis:

(i) A distributed OLTP database can support high scalability requirements by simply adopting an on-the-fly live data migration technique similar to the mobile subscribers *roaming* in a mobile wireless network.

(ii) The repartitioning KPIs—I_d, L_b, D_m—have explicit physical interpretations relating to the OLTP workload characteristics in terms of transactional recurrence and server span, physical data distributions, and CPU and I/O parallelism of the underlying database cluster.

(iii) By including the MNDTs that are incidental to a DT in a graph *min-cut* based repartitioning process, can prevent them turning into DTs in future.

(iv) Compact network representations are better for performing graph *min-cut* clustering as they can preserve more edges (by the contraction process) closely related to each other.

(v) Graph theoretic repartitioning can only provide sub-optimal results for reducing the impacts of DTs where load-balance comes as a by-product and there is no explicit control over data migrations.

(vi) The simple heuristic to prevent processing certain DTs in order to preserve the gains achieved by their already processed incident DTs, helps to avoid *ping-pong* effects during a repartitioning cycle. This is similar to the *thrashing* effect as well.

(vii) In greedy heuristic based incremental repartitioning it is possible to control the convergence of I_d by controlling the number of migration plans to be computed and calculating net I_d and L_b gains optimised per data migration.

(viii) The repartitioning KPIs I_d and L_b are not fully orthogonal to each other. Therefore, one can not be traded-off for another in a straightforward way.

(ix) The size of the transactional workload to be analysed can be significantly reduced by utilising stream mining to construct a compact and representative workload network which is much smaller to maintain and can be used for low granularity knowledge discovery.

(x) It is possible find an approximate repartitioning solution by directly applying the knowledge derived from transactional associativity with the representative workload network.

(xi) An atomic incremental repartitioning approach requires less computational footprints and the overall cost of repartitioning decision making and data migration operations can be amortised over the entire observation period.

(xii) A *TPC-C* workload shows high similarity to a random graph network with a high number of *hubs* with low node degree, therefore achieving repartitioning objectives is challenging and difficult to maintain over time.

(xiii) A *Twitter* workload contains a small number of *hubs* with a very high node degree, therefore repartitioning objectives are much easier to achieve and maintain.

7.2 Future Research Directions

With the rapid increase and growth of Internet-scale Web applications, workload-aware data management in shared-nothing distributed OLTP databases is a challenging task to achieve. Based on the research findings in this thesis we would like to point to the following open research challenges.

(i) In this thesis, we have observed the interesting dilemma of obtaining a *sub-optimal*, a *near-optimal*, and an *approximate* incremental repartitioning decision. An immediate extension of this research would have been a theoretical and mathematical analysis of these three unique decision choices based on our proposed repartitioning KPIs. This would help future researchers to understand and further analyse the inherent complexities of this problem domain with respect to proper physical interpretations.

(ii) The primary challenge remaining in a workload-aware incremental repartitioning framework is how we can analyse a snapshot of the workload network representation efficiently in order to make appropriate clustering decisions either at higher or lower granularity in order to quickly reduce the sever impacts of DTs. In this

context, a detailed analysis can be carried out from a network science perspective. One of the interesting ideas of colocating associative data tuples in the workload network would be by measuring the *assortativity* and *degree correlations*. This would assist the repartitioning scheme to identify the node connectivity patterns and answer questions such as whether the *hubs* are internally connected or they are avoiding each other, i.e., connected to many other *small-degree* nodes. From our observations in the thesis, it seems the *TPC-C* workload network may have a *disassortative* pattern where *hubs* are connected to many other *small-degree* nodes. In contrast, the *Twitter* workload network might exhibit an *assortativity* pattern where it seems the *hubs* are connected to each other and *small-degree* data tuples are connected to other such tuples.

(iii) From the network science context, another interesting approach would be using network *epidemic* and *contagion* analysis to understand how the *FCTs* and *semi-FCTs* manipulate the behaviour DTs in an OLTP environment. Such analysis would help us designing a *proactive* repartitioning scheme that might predict the pattern and potential impacts of DTs in large-scale OLTP system in advance.

(iv) Due to its small computational footprint, transactional stream mining based atomic incremental repartitioning can be further extended to identify *concept drifts* in the *ARH* network and its clustering. This important detection would tell us how frequently the repartitioning framework should be triggering the *ARHC* process to be more adaptive to the dynamic OLTP environment.

(v) Extension and improvement of the *OLTPDBSim* simulator can be another potential contribution to made by the research community. It is often challenging to implement diverse application scenarios and workloads to test the performance of a proposed repartitioning scheme. At present, most of the synthetic OLTP workload generators and benchmarking tools are built to measure engine specific database performances only in terms of their transaction processing capabilities. However, to understand the effects of an incremental repartitioning process based on our proposed KPIs, it is sufficient to use a more simplistic workload generation tool based on parameterised statistical configurations so as to emulate interesting *network* properties.

To this end, the study and analysis of the workload-aware incremental repartitioning of shared-nothing distributed OLTP databases is more challenging and intractable than we had anticipated at the beginning. As we dive deep into this research, the challenge seems intertwined with many interesting and complex elements that are not trivial to deal with. However, we believe that, with the aid of the proposed repartitioning schemes, it is possible to near optimally solve this problem. Furthermore, future research in this particular problem domain can certainly benefit from following our footsteps in this thesis.

Appendix A

Transaction Generation Model

Listing A.1: MATLAB source code for sensitivity analysis of the transaction generation model

```
s = RandStream('mt19937ar','Seed',1);
RandStream.setGlobalStream(s);
W = 3600;          % Observation window size; 1 h or 3600 s
p = 0.15;          % Probability of new transaction generation
R = 1;             % Transaction generation rate
eta = R*p;         % Average new transaction generation rate
N = W*24;          % Simulate for 24 h
nT = 0;            % Transaction id
T = [];            % Matrix to hold the generated transactions
alpha = 0.6;       % Exponential averaging weight for recurrence periods
uNmaxT = W*0.25;% Target number of unique transactions; 25% of 3600
uNmax = W;         % Unrestricted repetition window
uNmax = max(1, round(uNmaxT*(1 - (eta/uNmaxT)^2)))*W; % Restricted window

for i = 1:N
    if isempty(T) || rand(s) ≤ p
        nT = nT + 1;
        T = [T, -nT];
        Period(nT) = uNmax;
        Time(nT) = i;
    else
        [¬,idx] = unique(abs(T(1:i-1)),'last');
        [¬,idx2] = sort(idx,'descend');

        uT = abs(T(idx(idx2(1:min(length(idx),uNmax)))));
        n = randi(s,length(uT));
        T = [T,uT(n)];

        Period(uT(n)) = Period(uT(n))*alpha + (i-Time(uT(n)))*(1-alpha);
        Time(uT(n)) = i;
    end
    if i ≥ W
        unq = unique(abs(T((i-W+1):i)));
        unq_len(i-W+1) = length(unq);
    end
end
```

References

[1] "Cisco visual networking index: Global mobile data traffic forecast update 2014—2019 white paper," 2014, (last accessed 2016-10-31). [Online]. Available: http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/mobile-white-paper-c11-520862.html (in page 1)

[2] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663. (in pages 4, 54, and 75)

[3] "Vitess – scaling MySQL databases for large scale Web services," 2015, (last accessed 2016-10-31). [Online]. Available: https://github.com/youtube/vitess (in pages 5, 39, 65, and 69)

[4] D. Abadi, R. Agrawal, A. Ailamaki, M. Balazinska, P. A. Bernstein, M. J. Carey, S. Chaudhuri, J. Dean, A. Doan, M. J. Franklin, J. Gehrke, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, H. V. Jagadish, D. Kossmann, S. Madden, S. Mehrotra, T. Milo, J. F. Naughton, R. Ramakrishnan, V. Markl, C. Olston, B. C. Ooi, C. Ré, D. Suciu, M. Stonebraker, T. Walter, and J. Widom, "The beckman report on database research," *SIGMOD Record*, vol. 43, no. 3, pp. 61–70, Dec. 2014. (in pages 5 and 11)

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, Oct. 2007. (in page 6)

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage sys-

tem for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008. (in pages 6 and 44)

[7] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010. (in page 6)

[8] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, p. 7. (in pages 6, 39, and 41)

[9] D. Pritchett, "BASE: An ACID alternative," *ACM Queue*, vol. 6, no. 3, pp. 48–55, May 2008. (in pages 6 and 42)

[10] W. Vogels, "Eventually consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008. (in pages 6, 40, and 42)

[11] ——, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. (in page 6)

[12] M. Stonebraker, "The traditional RDBMS wisdom is (almost certainly) all wrong," 2013, (last accessed: 2016-10-31). [Online]. Available: http://slideshot.epfl.ch/play/suri_stonebraker (in page 7)

[13] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *Journal of Object Oriented Programing*, vol. 1, no. 3, pp. 26–49, Aug. 1988. (in page 19)

[14] R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, May. 2002. (in page 19)

[15] K. P. Birman, *Guide to Reliable Distributed Systems - Building High-Assurance Applications and Cloud-Hosted Services*, ser. Texts in computer science. Springer, 2012. (in pages 20 and 33)

[16] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," in *Proceedings of the 2012 ACM*

SIGMOD International Conference on Management of Data, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 61–72. (in pages 21, 57, 58, and 69)

[17] "A primer on database clustering architectures," 2016, (last accessed 2016-10-31). [Online]. Available: http://www.scaledb.com/pdfs/ArchitecturePrimer.pdf (in page 22)

[18] "Shared-disk vs. shared-nothing: Comparing architectures for clustered databases," 2016, (last accessed 2016-10-31). [Online]. Available: http://www.scaledb.com/pdfs/WP_SDvSN.pdf (in page 22)

[19] M. Nicola and M. Jarke, "Performance modeling of distributed and replicated databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 4, pp. 645–672, Jul./Aug. 2000. (in pages 22 and 36)

[20] M. Stonebraker, "The case for shared nothing," *IEEE Database Engineering Bulletin*, vol. 9, no. 1, pp. 4–9, 1986. (in page 23)

[21] B. G. Lindsay, "Jim gray at ibm: The transaction processing revolution," *SIGMOD Record*, vol. 37, no. 2, pp. 38–40, Jun. 2008. (in page 23)

[22] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. (in page 24)

[23] "Distributed Transaction Processing: The XA specification," 2016, (last accessed 2016-10-31). [Online]. Available: http://www2.opengroup.org/ogsys/catalog/c193 (in page 24)

[24] "Distributed Relational Database Architecture (DRDA) standard," 2016, (last accessed 2016-10-31). [Online]. Available: http://collaboration.opengroup.org/dbiop/ (in page 24)

[25] B. Kemme, R. J. Peris, and M. Patio-Martnez, *Database Replication*, 1st ed. Morgan and Claypool Publishers, 2010. (in page 26)

[26] "Wikipedia – Consensus (computer science)," 2016. [Online]. Available: http://en.wikipedia.org/wiki/Consensus_(computer_science) (in page 26)

[27] I. L. Traiger, J. Gray, C. A. Galtieri, and B. G. Lindsay, "Transactions and consistency in distributed database systems," *ACM Transactions on Database Systems (TODS)*, vol. 7, no. 3, pp. 323–342, Sep. 1982. (in page 27)

[28] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, no. 3, pp. 382–401, Jul. 1982. (in page 27)

[29] J. Gray, "Notes on database operating systems," in *Operating Systems, An Advanced Course*. London, UK, UK: Springer-Verlag, 1978, pp. 393–481. (in page 27)

[30] D. Skeen and M. Stonebraker, "A formal model of crash recovery in a distributed system," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 219–228, May 1983. (in page 28)

[31] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, Mar. 2006. (in pages 28 and 29)

[32] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998. (in pages 28 and 29)

[33] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, no. 11, pp. 624–633, Nov. 1976. (in page 30)

[34] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987. (in page 30)

[35] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, vol. 13, no. 2, pp. 185–221, Jun. 1981. (in page 31)

[36] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *Proceed-*

ings of the 8th USENIX conference on Networked systems design and implementation, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 11–26. (in page 32)

[37] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, Jul. 2003. (in page 33)

[38] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350. (in page 33)

[39] "Apache ZooKeeper," 2016, (last accessed 2016-10-31). [Online]. Available: http://zookeeper.apache.org/ (in page 33)

[40] "Consul by HasiCorp–service discovery and configuration made easy," 2016, (last accessed 2016-10-31). [Online]. Available: http://www.consul.io (in page 33)

[41] K. Birman, D. Malkhi, and R. V. Renesse, "Virtually synchronous methodology for dynamic service replication," Microsoft Research, Tech. Rep. MSR-TR-2010-151, Nov 2010. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=141727 (in page 33)

[42] "Isis^2—Cloud Computing Library," 2016, (last accessed 2016-10-31). [Online]. Available: http://isis2.codeplex.com (in pages 33 and 42)

[43] K. Birman, Q. Huang, and D. Freedman, "Overcoming the "d" in cap: Using isis2 to build locally responsive cloud services," Dept. of Computer Science; Cornell University, Ithaca NY 14850, Tech. Rep., April 2011. [Online]. Available: www.cs.cornell.edu/projects/quicksilver/public_pdfs/isis-socc.pdf (in page 33)

[44] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," *SIGMOD Record*, vol. 25, no. 2, pp. 173–182, Jun. 1996. (in pages 34 and 35)

[45] ——, "The dangers of replication and a solution," in *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '96. New York, NY, USA: ACM, 1996, pp. 173–182. (in page 35)

[46] M. Patiño Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Middle-r: Consistent database replication at the middleware level," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 4, pp. 375–423, Nov. 2005. (in page 38)

[47] R. Jimenez-Peris, M. Patiño Martinez, B. Kemme, F. Perez-Sorrosal, and D. Serrano, "Architecting dependable systems vi," in *A System of Architectural Patterns for Scalable, Consistent and Highly Available Multi-Tier Service-Oriented Infrastructures*, R. Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M. Beek, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 1–23. (in pages 38 and 39)

[48] F. Perez-Sorrosal, M. Patiño Martinez, R. Jimenez-Peris, and B. Kemme, "Consistent and scalable cache replication for multi-tier J2EE applications," in *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 328–347. (in page 39)

[49] ——, "Consistent and scalable cache replication for multi-tier J2EE applications," in *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, ser. MIDDLEWARE2007. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 328–347. (in page 39)

[50] F. Perez-Sorrosal, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Elastic SI-Cache: consistent and scalable caching in multi-tier architectures," *The VLDB Journal*, vol. 20, no. 6, pp. 841–865, Dec. 2011. (in page 39)

[51] "MySQL toolkit for managing billions of rows and hundreds of database machines," 2016, (last accessed 2016-10-31). [Online]. Available: https://github.com/tumblr/jetpants (in pages 39, 65, and 69)

[52] "A flexible sharding framework for creating eventually-consistent distributed datastores," 2016, (last accessed 2016-10-31). (in pages 39, 65, and 69)

[53] "Wikipedia – NoSQL," 2016, (last accessed 2016-10-31). [Online]. Available: http://en.wikipedia.org/wiki/NoSQL (in page 39)

[54] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant Web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002. (in pages 39 and 41)

[55] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009, (last accessed 2016-10-31). [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html (in page 40)

[56] E. Brewer, "CAP twelve years later: How the "rules" have changed," *IEEE Computer*, vol. 45, no. 2, pp. 23–29, Feb. 2012. (in page 41)

[57] K. Birman, D. Freedman, Q. Huang, and P. Dowell, "Overcoming CAP with consistent soft-state replication," *IEEE Computer*, vol. 45, no. 2, pp. 50–58, Feb. 2012. (in page 42)

[58] D. J. Abadi, "Problems with CAP, and Yahoo's little known NoSQL system," April 2010. [Online]. Available: http://dbmsmusings.blogspot.com.au/2010/04/problems-with-cap-and-yahoos-little.html (in page 43)

[59] ——, "CAP, PACELC, and Determinism," Jan. 2011. [Online]. Available: http://www.slideshare.net/abadid/cap-pacelc-and-determinism (in page 43)

[60] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach, "SW-Store: a vertically partitioned DBMS for Semantic Web data management," *The VLDB Journal*, vol. 18, no. 2, pp. 385–406, Apr. 2009. (in page 43)

[61] "Apache Cassandra Project," 2016, (last accessed 2016-10-31). [Online]. Available: http://cassandra.apache.org (in pages 43, 47, and 60)

[62] "Amazon DynamoDB — NoSQL cloud database service," 2016, (last accessed

2016-10-31). [Online]. Available: http://aws.amazon.com/dynamodb (in page 43)

[63] "Riak KV — Distributed NoSQL Database," 2016, (last accessed 2016-10-31). [Online]. Available: http://basho.com/products/riak-kv/ (in page 43)

[64] "MongoDB," 2016, (last accessed 2016-10-31). [Online]. Available: http://www.mongodb.org/ (in pages 44 and 47)

[65] "Apache CouchDB," 2016, (last accessed 2016-10-31). [Online]. Available: http://couchdb.apache.org (in page 44)

[66] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, Aug. 2008. (in page 44)

[67] "Apache HBase," 2016, (last accessed 2015-07-24). [Online]. Available: http://hbase.apache.org (in pages 44 and 47)

[68] "H-Store – Next Generation OLTP Database Research," 2016, (last accessed: 2016-10-31). [Online]. Available: http://hstore.cs.brown.edu (in pages 44, 57, and 69)

[69] "VoltDB: In-Memory Database, NewSQL & Real-Time Analytics," 2016, (last accessed 2016-10-31). [Online]. Available: http://voltdb.com/ (in pages 44, 47, and 63)

[70] D. Weinreb, "Improving the PACELC taxonomy," Jan. 2011. [Online]. Available: http://danweinreb.org/blog/improving-the-pacelc-taxonomy (in page 44)

[71] R. Ramakrishnan, "CAP and cloud data management," *Computer, IEEE*, vol. 45, no. 2, pp. 43–49, Feb. 2012. (in page 45)

[72] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. (in page 45)

[73] M. Stonebraker, "Errors in database systems, eventual consistency, and the cap theorem," *Blog, Communications of the ACM*, Apr. 2010. [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext (in page 46)

[74] "MySQL Cluster CGE," 2016, (last accessed 2015-07-24). [Online]. Available: http://www.mysql.com/products/cluster/ (in page 47)

[75] "Google Cloud SQL – a fully-managed relational MySQL database service," 2016, (last accessed 2016-10-31). [Online]. Available: http://cloud.google.com/sql/ (in page 47)

[76] "SQL Database – Relational Database Service – Microsoft Azure," 2016, (last accessed 2015-07-24). [Online]. Available: http://azure.microsoft.com/en-us/services/sql-database/ (in page 47)

[77] J. Waldo, "Scaling in games and virtual worlds," *Communications of the ACM*, vol. 51, no. 8, pp. 38–44, Aug. 2008. (in page 48)

[78] M. Kohana, S. Okamoto, M. Kamada, and T. Yonekura, "Dynamic data allocation scheme for multi-server Web-based morpg system," in *Proceedings of the 2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, ser. WAINA '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 449–454. (in page 48)

[79] ——, "Dynamic reallocation rules on multi-server Web-based morpg system," *International Journal of Grid and Utility Computing*, vol. 3, no. 2/3, pp. 136–144, Jul. 2012. (in page 48)

[80] A. Floratou, N. Teletia, D. J. DeWitt, J. M. Patel, and D. Zhang, "Can the elephants handle the nosql onslaught?" *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1712–1723, Aug. 2012. (in page 48)

[81] L. R. Ford Jr. and D. R. Fulkerson, "Maximal flow through a network," *Maximal flow through a network*, vol. 4, pp. 399–404, 1956. (in page 52)

[82] "Wikipedia – Minimum k-cut," 2016, (last accessed 2016-10-31). [Online]. Available: http://en.wikipedia.org/wiki/Minimum_k-cut (in page 52)

[83] D. R. Karger, "Global min-cuts in rnc, and other ramifications of a simple min-out algorithm," in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '93. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1993, pp. 21–30. (in page 52)

[84] G. Karypis and V. Kumar, "Parallel multilevel k-way partitioning scheme for irregular graphs," in *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996. (in page 52)

[85] ——, "Multilevel k-way hypergraph partitioning," in *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, ser. DAC '99. New York, NY, USA: ACM, 1999, pp. 343–348. (in pages 52, 111, 113, and 170)

[86] "Wikipedia – Matching (graph theory)," 2016, (last accessed 2016-10-31). [Online]. Available: https://en.wikipedia.org/wiki/Matching_(graph_theory) (in page 52)

[87] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *Journal of Parallel and Distributed Computing*, vol. 48, no. 1, pp. 96–129, Jan. 1998. (in pages 52, 111, and 113)

[88] C. Schulz, "Scalable parallel refinement of graph partitions," 2009, (last accessed 2016-10-31). [Online]. Available: http://algo2.iti.kit.edu/documents/DAchristianschulz.pdf (in page 52)

[89] "METIS – serial graph partitioning and fill-reducing matrix ordering," 2016, (last accessed 2016-10-31). [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview (in pages 52, 56, 61, 111, and 113)

[90] "hMETIS – hypergraph & circuit partitioning," 2016, (last accessed 2016-10-31). [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview (in pages 52, 61, 111, 113, 170, and 173)

[91] A. Tatarowicz, C. Curino, E. Jones, and S. Madden, "Lookup tables: Fine-grained partitioning for distributed databases," in *IEEE 28th International Conference on Data Engineering (ICDE)*, April 2012, pp. 102–113. (in page 53)

[92] N. Xu, "Fine-grained data partitioning framework for distributed database systems," in *Proceedings of the Companion Publication of the 23rd International Conference on World Wide Web Companion*, ser. WWW Companion '14. Republic and Canton of Geneva, Switzerland: International World Wide Web Conferences Steering Committee, 2014, pp. 57–62. (in pages 53 and 69)

[93] K. Kumar, A. Quamar, A. Deshpande, and S. Khuller, "Sword: workload-aware data placement and replica selection for cloud data management systems," *The VLDB Journal*, pp. 1–26, 2014. (in pages 53, 63, 69, 92, and 148)

[94] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Looking up data in p2p systems," *Communications of the ACM*, vol. 46, no. 2, pp. 43–48, Feb. 2003. (in page 54)

[95] M. Mehta and D. J. DeWitt, "Data placement in shared-nothing parallel database systems," *The VLDB Journal*, vol. 6, no. 1, pp. 53–72, Feb. 1997. (in page 55)

[96] S. Agrawal, V. Narasayya, and B. Yang, "Integrating vertical and horizontal partitioning into automated physical database design," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, pp. 359–370. (in page 55)

[97] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "Db2 design advisor: Integrated automatic physical database design," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB '04. VLDB Endowment, 2004, pp. 1087–1097. (in page 55)

[98] R. Nehme and N. Bruno, "Automated partitioning design in parallel database systems," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1137–1148. (in pages 55 and 69)

[99] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 981–992. (in page 56)

[100] S. Das, "Scalable and elastic transactional data stores for cloud computing platforms," Ph.D. dissertation, Department of Computer Science, University of California, Santa Barbara, Sept. 2011. [Online]. Available: http://www.cs.ucsb.edu/~sudipto/diss/ (in pages 56, 58, and 59)

[101] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, Sep. 2010. (in pages 56, 58, 69, 92, 102, 106, and 113)

[102] C. Curino, E. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich, "Relational cloud: A database service for the cloud," in *5th Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, Jan. 2011. (in pages 56, 58, and 69)

[103] E. P. C. Jones, "Fault-tolerant distributed transactions for partitioned oltp databases," Phd Thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, Feb. 2012, (last accessed 2016-10-31). [Online]. Available: http://hdl.handle.net/1721.1/71477 (in pages 58 and 69)

[104] A. Turcu, R. Palmieri, and B. Ravindran, "Automated data partitioning for independent distributed transactions," in *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, ser. MiddlewareDPT '13. New York, NY, USA: ACM, 2013, pp. 11:1–11:2. (in page 58)

[105] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 21–21. (in page 58)

[106] S. Das, D. Agrawal, and A. El Abbadi, "G-Store: a scalable data store for transactional multi key access in the cloud," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 163–174. (in page 59)

[107] ——, "ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud," *ACM Transactions on Database Systems*, vol. 38, no. 1, pp. 5:1–5:45, Apr. 2013. (in pages 59 and 69)

[108] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 494–505, May 2011. (in page 59)

[109] A. J. Elmore, S. Das, D. Agrawal, and A. El Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD'11. New York, NY, USA: ACM, 2011, pp. 301–312. (in page 59)

[110] A. J. Elmore, "Elasticity primitives for database as a service," Phd Thesis, University Of California, Santa Barbara, 2014, (last accessed 2016-10-31). [Online]. Available: http://cs.ucsb.edu/~aelmore/ElmoreDis.pdf (in pages 59, 65, and 69)

[111] A. J. Elmore, V. Arora, R. Taft, A. Pavlo, D. Agrawal, and A. El Abbadi, "Squall: Fine-grained live reconfiguration for partitioned main memory databases," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 299–313. (in pages 59 and 69)

[112] K. Chen, Y. Zhou, and Y. Cao, "Scheduling online repartitioning in oltp systems," in *Proceedings of the Middleware Industry Track*, ser. Industry papers. New York, NY, USA: ACM, 2014, pp. 4:1–4:6. (in pages 59 and 69)

[113] ——, "Online data partitioning in distributed database systems," in *Proceedings*

of the 18th International Conference on Extending Database Technology, ser. EDBT '15. NY, USA: ACM, 2015. (in pages 59 and 69)

[114] J. M. Pujol, "Scaling online social networks without pains," in *5th International Workshop on Networking Meets Databases, NetDB'09*, ser. NetDB'09, 2009. (in page 60)

[115] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez, "The little engine(s) that could: scaling online social networks," *IEEE/ACM Trans. Networking*, vol. 20, no. 4, pp. 1162–1175, Aug. 2012. (in pages 60 and 69)

[116] A. Turk, R. O. Selvitopi, H. Ferhatosmanoglu, and C. Aykanat, "Temporal workload-aware replicated partitioning for social networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 11, pp. 2832–2845, Nov. 2014. (in pages 60 and 106)

[117] M. Thomae and O. Ricardo, "Database partitioning strategies for social network data," Master's thesis, Massachusetts Institute of Technology, USA, 2012, (last accessed 2016-10-31). [Online]. Available: http://dspace.mit.edu/handle/1721.1/77449 (in pages 60 and 61)

[118] "Zipf distribution – from wolfram mathworld," 2016, (last accessed 2016-10-31). [Online]. Available: http://mathworld.wolfram.com/ZipfDistribution.html (in pages 61, 83, 84, and 153)

[119] C. Yu, G. Xiaoyan, and T. Stephen, "Hyper-graph based database partitioning for transactional workloads," *Computing Research Repository (CoRR)*, vol. abs/1309.1556, 2013, (last accessed 2016-10-31). [Online]. Available: http://arxiv.org/abs/1309.1556 (in pages 61 and 69)

[120] C. Yu, G. Xiaoyan, Z. Baoyao, and S. Todd, "HOPE: Iterative and interactive database partitioning for OLTP workloads," in *IEEE 30th International Conference on Data Engineering (ICDE)*, March 2014, pp. 1274–1277. (in pages 61 and 69)

[121] C. Yu, G. Xiaoyan, and T. Stephen, "Database partitioning for data processing system," May 2015, (last accessed 2016-10-31). [Online]. Available: http://www.google.com/patents/US9031994 (in pages 61 and 69)

[122] Z. Shang and Y. J.X., "Catch the wind: Graph workload balancing on cloud," in *IEEE 29th International Conference on Data Engineering (ICDE)*, April 2013, pp. 553–564. (in pages 62 and 69)

[123] T. Rafiq, "Elasca: Workload-aware elastic scalability for partition based database systems," Master's thesis, University of Waterloo, Canada, May 2013, (last accessed 2016-10-31). [Online]. Available: http://uwspace.uwaterloo.ca/handle/10012/7525 (in pages 62 and 69)

[124] K. A. Kumar, A. Deshpande, and K. Samir, "Data placement and replica selection for improving co-location in distributed environments," *Computing Research Repository (CoRR)*, vol. abs/1302.4168, 2013, (last accessed 2016-10-31). [Online]. Available: http://arxiv.org/abs/1302.4168 (in pages 63 and 148)

[125] A. Quamar, K. A. Kumar, and A. Deshpande, "SWORD: scalable workload-aware data placement for transactional workloads," in *Proceedings of the 16th International Conference on Extending Database Technology*, ser. EDBT '13. NY, USA: ACM, 2013, pp. 430–441. (in pages 63, 102, 106, 107, and 148)

[126] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 245–256, Nov. 2014. (in pages 64 and 69)

[127] "Wikipedia – Bin packing problem," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Bin_packing_problem (in page 64)

[128] "Wikipedia – Hockey Stick Graph," 2016, (last accessed: 2016-10-31). [Online]. Available: http://en.wikipedia.org/wiki/Hockey_stick_graph (in page 65)

[129] "Wikipedia – NewSQL," 2016. [Online]. Available: https://en.wikipedia.org/wiki/NewSQL (in page 65)

[130] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas, "Accordion: Elastic scalability for database systems supporting distributed transactions," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1035–1046, Aug. 2014. (in pages 65 and 69)

[131] "Vitess: Scaling mysql at youtube using go," 2015, (last accessed 2016-10-31). [Online]. Available: http://www.usenix.org/conference/lisa12/vitess-scaling-mysql-youtube-using-go (in page 65)

[132] "MySQL Fabric," 2016, (last accessed 2016-10-31). [Online]. Available: www.mysql.com/products/enterprise/fabric.html (in pages 66 and 69)

[133] C. Curino, E. P. Jones, S. Madden, and H. Balakrishnan, "Workload-aware database monitoring and consolidation," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 313–324. (in page 69)

[134] A. Pavlo, E. P. C. Jones, and S. Zdonik, "On predictive modeling for optimizing transaction execution in parallel oltp systems," *Proceedings of the VLDB Endowment*, vol. 5, no. 2, pp. 85–96, Oct. 2011. (in page 69)

[135] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, Aug. 2008. (in page 69)

[136] G. Roth, "Server load balancing architectures–high scalability and availability for server farms," Oct 2008, (last accessed 2016-10-31). [Online]. Available: http://www.javaworld.com/article/2077921/architecture-scalability/server-load-balancing-architectures--part-1--transport-level-load-balancing.html (in page 72)

[137] "Memcached – high-performance, distributed memory object caching system," 2016, (last accessed 2016-10-31). [Online]. Available: http://memcached.org (in page 72)

[138] "Redis – in-memory data structure store, used as database, cache, and message broker," 2016, (last accessed 2016-10-31). [Online]. Available: http://redis.io/ (in page 72)

[139] "Eclipselink," 2016, (last accessed 2016-10-31). [Online]. Available: http://www.eclipse.org/eclipselink/ (in page 73)

[140] "Apache openjpa," 2015, (last accessed 2016-10-31). [Online]. Available: http://openjpa.apache.org/ (in page 73)

[141] "Wikipedia – Roaming in GSM Network," 2016, (last accessed 2016-10-31). [Online]. Available: http://en.wikipedia.org/wiki/Roaming (in page 74)

[142] "Wikipedia – Mobile IP," 2016, (last accessed 2016-10-31). [Online]. Available: http://en.wikipedia.org/wiki/Mobile_IP (in page 75)

[143] "Wikipedia – Care-of address," 2016, (last accessed 2016-10-31). [Online]. Available: https://en.wikipedia.org/wiki/Care-of_address (in page 75)

[144] "SSJ: Stochastic Simulation in Java," 2016, (last accessed 2016-10-31). [Online]. Available: http://simul.iro.umontreal.ca/ssj/indexe.html (in page 80)

[145] "TPC-C – On-line Transaction Processing Benchmark," 2016, (last accessed 2016-10-31). [Online]. Available: http://www.tpc.org/tpcc/ (in page 82)

[146] "OLTPBenchmark Website," 2016, (last accessed 2016-10-31). [Online]. Available: http://oltpbenchmark.com/ (in pages 83 and 85)

[147] D. Krishnamurthy, "Synthetic workload generation for stress testing session-based systems," Ph.D. dissertation, Department of Systems and Computer Engineering, Ottawa, ON, Canada, 2004. [Online]. Available: https://curve.carleton.ca/90781bb3-18fd-49e8-a8db-3acbc50b1616 (in page 84)

[148] D. Krishnamurthy, J. Rolia, and S. Majumdar, "A synthetic workload generation technique for stress testing session-based systems," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 868–882, Nov. 2006. (in page 84)

[149] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, S. Patil, A. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for Web 2.0," in *1st Workshop on Cloud Computing and Its Applications, October 2008. CCA 2008*, 2008. (in page 84)

[150] "Faban harness and benchmark framework," 2016, (last accessed: 2016-10-31). [Online]. Available: https://java.net/projects/faban/ (in page 84)

[151] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 37–48, Mar. 2012. (in page 85)

[152] "Cloudsuite 2.0: a benchmark suite for emerging scale-out applications," 2016, (last accessed: 2016-10-31). [Online]. Available: http://parsa.epfl.ch/cloudsuite/cloudsuite.html (in page 85)

[153] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson, "Rain: A workload generation toolkit for cloud computing applications," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-14, Feb. 2010, (last accessed 2016-10-31). [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-14.html (in page 85)

[154] R. Griffith, "Rain workload toolkit," 2016, (last accessed: 2016-10-31). [Online]. Available: https://github.com/rean/rain-workload-toolkit (in page 85)

[155] C. A. Curino, D. E. Difallah, A. Pavlo, and P. Cudre-Mauroux, "Benchmarking OLTP/Web databases in the cloud: The oltp-bench framework," in *Proceedings of the Fourth International Workshop on Cloud Data Management*, ser. CloudDB '12. New York, NY, USA: ACM, 2012, pp. 17–20. (in page 85)

[156] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 4, pp. 277–288, Dec. 2013. (in page 85)

[157] N. J. Gunther, "A simple capacity model of massively parallel transaction systems," in *Proceedings of the 1993 CMG Conference*. San Diego, CA, USA: Computer Measurement Group, 1993, pp. 1035–1044. (in page 91)

[158] A.-L. Barabási, R. Albert, and H. Jeong, "Scale-free characteristics of random networks: the topology of the World Wide Web," *Physica A: Statistical Mechanics and its Applications*, vol. 281, no. 1—4, pp. 69–77, 2000. (in page 135)

[159] I. Yang, H. Jeong, B. Kahng, and A.-L. Barabási, "Emerging behavior in electronic bidding," *The Physical Review Journals E*, vol. 68, p. 016102, Jul 2003. (in page 135)

[160] A.-L. Barabási, "The origin of bursts and heavy tails in human dynamics," *Nature*, vol. 435, 2005. (in page 135)

[161] Z. Dezsö, E. Almaas, A. Lukács, B. Rácz, I. Szakadát, and A.-L. Barabási, "Dynamics of information access on the Web," *The Physical Review Journals E*, vol. 73, p. 066132, Jun 2006. (in page 135)

[162] L. Backstrom, "Anatomy of Facebook," 2011, (last accessed 2016-10-31). [Online]. Available: https://www.facebook.com/notes/facebook-data-team/anatomy-of-facebook/10150388519243859 (in page 135)

[163] S. Milgram, "The small world problem," *Psychology Today*, vol. 2, pp. 60–67, 1967. (in page 135)

[164] T. Jeffrey and S. Milgram, "An experimental study of the small world problem," *Sociometry*, vol. 32, no. 4, pp. 425–443, 1969. (in page 135)

[165] G. Cormode and M. Hadjieleftheriou, "Methods for finding frequent items in data streams," *The VLDB Journal*, vol. 19, no. 1, pp. 3–20, Feb. 2010. (in pages 153, 156, and 158)

[166] A. Bifet, "Adaptive stream mining: Pattern learning and mining from evolving data streams," in *Proceedings of the 2010 Conference on Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams*. Amsterdam, The Netherlands: IOS Press, 2010, pp. 1–212. (in pages 153 and 158)

[167] G. Cormode, F. Korn, S. Muthukrishnan, and D. Srivastava, "Finding hierarchical heavy hitters in streaming data," *ACM Transactions on Knowledge Discovery from Data*, vol. 1, no. 4, pp. 2:1–2:48, Feb. 2008. (in page 153)

[168] E.-H. S. Han, G. Karypis, V. Kumar, and B. Mobasher, "Clustering based on association rule hypergraphs," in *Proceedings of the ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD'97)*. ACM Press, 1997, pp. 9–13. (in pages 154 and 170)

[169] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 346–357. (in page 156)

[170] G. Cormode and S. Muthukrishnan, "What's hot and what's not: Tracking most frequent items dynamically," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 249–278, Mar. 2005. (in page 156)

[171] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," *Proceedings of the VLDB Endowment*, vol. 5, no. 12, pp. 1699–1699, Aug. 2012. (in page 156)

[172] "MOA – Massive Online Analysis," 2016, (last accessed 2016-10-31). [Online]. Available: http://moa.cms.waikato.ac.nz (in pages 156 and 160)

[173] "RapidMiner – Open Source Predictive Analytics Platform," 2016, (last accessed 2016-10-31). [Online]. Available: https://rapidminer.com (in page 156)

[174] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. (in page 157)

[175] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Records*, vol. 29, no. 2, pp. 1–12, May 2000. (in page 157)

[176] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *Proceedings of the 7th International Conference*

on Database Theory, ser. ICDT '99. London, UK, UK: Springer-Verlag, 1999, pp. 398–416. (in page 158)

[177] M. J. Zaki, "Generating non-redundant association rules," in *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '00, 2000, pp. 34–43. (in page 158)

[178] M. J. Zaki and C.-J. Hsiao, "CHARM: An efficient algorithm for closed itemset mining," in *Second SIAM International Conference on Data Mining, SDM*, 2002, pp. 457—473. (in page 158)

[179] J. Pei, J. Han, and R. Mao, "CLOSET: An efficient algorithm for mining frequent closed itemsets," in *Proceedings of the 2000 ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, 2000, pp. 21–30. (in page 158)

[180] C. Giannella, J. Han, J. Pei, X. Yan, and P. S. Yu, "Mining frequent patterns in data streams at multiple time granularities," in *Proceedings of the NSF Workshop on Next Generation Data Mining*, 2002. (in page 158)

[181] Y. Chi, H. Wang, P. S. Yu, and R. R. Muntz, "Catch the moment: Maintaining closed frequent itemsets over a data stream sliding window," *Knowledge and Information Systems*, vol. 10, no. 3, pp. 265–294, Oct. 2006. (in page 158)

[182] H.-F. Li, C.-C. Ho, and S.-Y. Lee, "Incremental updates of closed frequent itemsets over continuous data streams," *Expert Systems with Applications*, vol. 36, no. 2, Part 1, pp. 2451 – 2458, 2009. (in page 158)

[183] D. Ienco, A. Bifet, I. Žliobaitė, and B. Pfahringer, "Clustering based active learning for evolving data streams," in *Discovery Science*, ser. Lecture Notes in Computer Science, J. Fürnkranz, E. Hüllermeier, and T. Higuchi, Eds. Springer, 2013, vol. 8140, pp. 79–93. (in page 158)

[184] J. a. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Survey*, vol. 46, no. 4, pp. 44:1–44:37, Mar. 2014. (in page 158)

[185] J. Cheng, Y. Ke, and W. Ng, "Maintaining frequent closed itemsets over a sliding window," *Journal of Intelligent Information Systems*, vol. 31, no. 3, pp. 191–215, 2008. (in pages 158 and 160)

[186] M. J. Zaki and C.-J. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," *IEEE Transaction on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 462–478, Apr. 2005. (in page 160)

[187] M. Quadrana, A. Bifet, and R. Gavaldà, "An efficient closed frequent itemset miner for the moa stream mining system," *AI Communications*, vol. 28, no. 1, pp. 143–158, 2015. (in page 160)

[188] D. Boley, M. Gini, R. Gross, E.-H. S. Han, K. Hastings, G. Karypis, V. Kumar, B. Mobasher, and J. Moore, "Partitioning-based clustering for Web document categorization," *Decision Support Systems*, vol. 27, no. 3, pp. 329–341, 1999. (in page 170)

[189] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa, "Discovery and evaluation of aggregate usage profiles for web personalization," *Data Mining and Knowledge Discovery*, vol. 6, no. 1, pp. 61–82, 2002. (in page 170)

Index