Future Generation Computer Systems 56 (2016) 421–435

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable OLTP applications

Joarder Kamal[a,*], Manzur Murshed[b], Rajkumar Buyya[c]

[a] *Faculty of Information Technology, Monash University, Australia*

[b] *Faculty of Science and Technology, Federation University Australia, Australia*

[c] *Cloud Computing and Distributed Systems (CLOUDS) Laboratory, Department of Computing and Information Systems, The University of Melbourne, Australia*

HIGHLIGHTS

- We propose incremental repartitioning of distributed OLTP databases for high-scalability.
- We model two incremental repartitioning algorithm and lookup mechanism.
- We develop a unique transaction generation model for simulation.
- We derive novel impact metrics for distributed transactions.
- Simulation results indicate adaptability of the methods scalable OLTP applications.

ARTICLE INFO

Article history:
Received 3 March 2015
Received in revised form
18 August 2015
Accepted 21 September 2015
Available online 3 October 2015

Keywords:
Shared-nothing
OLTP
Distributed transactions
Incremental repartitioning
Load-balance
Data migration

ABSTRACT

On-line Transaction Processing (OLTP) applications often rely on shared-nothing distributed databases that can sustain rapid growth in data volume. Distributed transactions (DTs) that involve data tuples from multiple geo-distributed servers can adversely impact the performance of such databases, especially when the transactions are short-lived and these require immediate responses. The *k*-way min-cut graph clustering based database repartitioning algorithms can be used to reduce the number of DTs with acceptable level of load balancing. Web applications, where DT profile changes over time due to dynamically varying workload patterns, frequent database repartitioning is needed to keep up with the change. This paper addresses this emerging challenge by introducing incremental repartitioning. In each repartitioning cycle, DT profile is learnt online and *k*-way min-cut clustering algorithm is applied on a special sub-graph representing all DTs as well as those non-DTs that have at least one tuple in a DT. The latter ensures that the min-cut algorithm minimally reintroduces new DTs from the non-DTs while maximally transforming existing DTs into non-DTs in the new partitioning. Potential load imbalance risk is mitigated by applying the graph clustering algorithm on the finer logical partitions instead of the servers and relying on random one-to-one cluster-to-partition mapping that naturally balances out loads. Inter-server data-migration due to repartitioning is kept in check with two special mappings favouring the current partition of majority tuples in a cluster—the many-to-one version minimising data migrations alone and the one-to-one version reducing data migration without affecting load balancing. A distributed data lookup process, inspired by the roaming protocol in mobile networks, is introduced to efficiently handle data migration without affecting scalability. The effectiveness of the proposed framework is evaluated on realistic TPC-C workloads comprehensively using graph, hypergraph, and compressed hypergraph representations used in the literature. To compare the performance of any incremental repartitioning framework without any bias of the external min-cut algorithm due to graph size variations, a transaction generation model is developed that can maintain a target number of unique transactions in any arbitrary observation window, irrespective of new transaction arrival rate. The overall impact of DTs at any instance is estimated from the exponential moving average of the recurrence period of unique transactions to avoid transient fluctuations. The effectiveness and adaptability of the proposed incremental repartitioning framework for transactional workloads have been established with extensive simulations on both range partitioned and consistent hash partitioned databases.

* Corresponding author. Tel.: +61 351226133; fax: +61 399055159.
E-mail addresses: joarder.kamal@monash.edu (J. Kamal), manzur.murshed@federation.edu.au (M. Murshed), rbuyya@unimelb.edu.au (R. Buyya).

http://dx.doi.org/10.1016/j.future.2015.09.024
0167-739X/© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Electronic data management was never more challenging than today with the global expansion of e-commerce, digital media, telecommunications, and social networks. It is estimated that around 2.3 trillion gigabytes of digitised data are generated everyday [1] with dynamic usage patterns during our daily interactions in the Web. As an example, we interact with over 30 billion pieces of shared contents (e.g., Web links, news stories, posts, notes, photos, etc.) in Facebook and watch more than 4 billion hours of videos in YouTube every month [1]. Such interactive Web traffic are primarily driven by On-line Transaction Processing (OLTP) applications requiring real-time responsiveness from end-users' perspective. From data management perspective, these applications generate short-lived transactions in underlying databases accessing less than hundred records—tuples from a fewer number of tables. OLTP applications typically require a normalised database schema, expose entity–relationship data model, and tend to scale-out for millions of simultaneously connected Web users. However, legacy Relational Database Management Systems (RDBMS) deployed in a *shared-nothing* distributed cluster serving modern OLTP workloads struggle to meet such high-scalability requirements.

Since the rise of cluster computing in late 90s, *shared-nothing* architecture triumphed over *shared-disk* systems for leveraging the power of commodity workstations and operating systems to scale-out Web applications. With the recent advancements of Cloud computing, *shared-nothing* distributed databases with horizontal data partitioning (also called *Sharding*) becomes the de-facto technique to manage scalable OLTP applications for the Web. Unfortunately, traditional horizontal data partitioning hardly adopt—dynamic workload characteristics, eliminate data hotspots, and sudden workload spikes without expansive data redistributions risking potential service downtimes within a geo-distributed database cluster [2]. Therefore, scaling out modern OLTP applications in a *shared-nothing* database cluster is extremely challenging. Transactional workloads are update intensive, highly concurrent, and require results within very low response times from the databases. The main bottleneck to scalability comes from concurrent nature of accessing shared application data structures and entity–relationships in a partitioned databases [3]. Distributed transactions (DTs) that access data tuples from multiple physical servers holding partitioned database tables adversely impact the database scalability while executing 2PC (Two-Phase Commit) and 2PL (Two-Phase Locking) operations to maintain ACID guarantees [4]. The only way to minimise the impact of DTs is to cluster the most frequently accessed data tuples together depending on the workload characteristics which requires dynamic repartitioning of the database.

Clustering frequently accessed workload tuples for eliminating DTs causes data distribution imbalance within the *shared-nothing* data nodes, requires expansive inter-node data migrations, and challenges scalable data lookup operations. Moreover, it is even difficult to pinpoint when we need to execute a repartitioning operation and whether in a proactive or reactive manner. Nevertheless, inappropriate and static partitioning can lead to unscalable transaction processing, potential disk failures or downtimes, and overloading inter-node bandwidth. In summary, the primary challenges to manage scalable OLTP applications deployed in a *shared-nothing* RDBS cluster are (1) how to incremental repartitioning to reduce the impact of DTs which arise from workload characteristics; (2) how to repartitioning the database independent of the number of physical servers; (3) how to perform on-the-fly data migration and minimise its cost; (4) how to perform scalable data lookup; and (5) how to maintain balance on server- and partition-level data distributions.

Recently proposed techniques for workload-aware data partitioning [5,6] monitor the transactional logs and periodically create workload networks using graph or hyper graph representation. Each edge in a workload graph connects a pair of tuples originated from the same transaction whereas a hyper edge connects all tuples within a transaction in a hypergraph. Later, these workload representations are clustered using *k*-way min-cut clustering, and then randomly placed across the set of physical servers within a database cluster. As long as workload characteristics do not change dramatically, and tuples from a cluster stay together in a physical server, the occurrences and adverse impacts of DTs are reduced rapidly. A number of centralised data lookup and routing mechanisms are also proposed to support such dynamic data redistribution. Large-scale OLTP service providers also utilise partition management libraries like YouTube's Vitess [7], Tumblr's JetPants [8], Twitter's Gizzard [9], and Apache Giraf [10] to deal with scalable data growth. Nonetheless, the underlying data placement techniques are not transparent to application codes, and redistributions are not aware of workload dynamics. Furthermore, none of these techniques provide any explicit way to minimise physical data migrations over WAN, and global load-balance at the same time for a geo-distributed *shared-nothing* cluster.

This paper presents a novel workload-aware incremental database repartitioning framework, which transparently redistributes tuples to reduce the overall impact of DTs while ensuring minimum data migrations and preserving global load-balance. The framework periodically collects and pre-processes transactional logs. A unique transaction classification process then identifies DTs and *moveable* non-DTs that share at least one tuple with a DT. A workload network is constructed using only these two types of transactions. A graph min-cut based clustering algorithm is used to create balanced clusters of tuples. Finally, these clusters are assigned to database partitions using a cluster-to-partition mapping algorithm. We perform sensitivity analysis by representing the workload networks in fine, exact, and coarse granularity using graphs, hypergraphs, and compressed hypergraphs. In contrary to previous works, a fixed number of clusters are created from the workload network for the total number of logical data partitions in the entire database instead of the number of physical servers. This provides finer control in load-balance over the set of both partitions and servers. We also avoid tuple-level replications to observe the quality of incremental repartitioning under worst-case scenario of DTs. We also propose two innovative cluster-to-partition mapping strategies that cater for minimising both physical data migrations and distribution imbalance. Our distributed data lookup mechanism ensures high-scalability, and guarantees a maximum of two lookups to locate a tuple within the partitioned database.

A set of novel metrics are developed to evaluate the quality of incremental repartitioning and also provide a way to administratively configure a particular repartitioning objective using a composite metric. We also design and develop a novel transaction generation model that is capable of simulating a wide-range of OLTP systems via parametric configurations. By enforcing the system to generate a fixed amount of unique transactions per observation window we carefully eliminate any undue effect of external graph or hypergraph partitioning libraries. We examine *reactive—hourly* and *proactive—threshold*-based incremental repartitioning strategies deployed in a *range* and a *consistent-hash* partitioned *shared-nothing* database cluster. We compare the results against a *no repartitioning* and a *static* repartitioning (similar to [5]) strategies for different workload representations and cluster-to-partition mapping techniques.

The main contributions are summarised in below:

1. Investigate possible design choices for workload network representations and their applicability.

2. Propose a proactive transaction classification technique that identifies DTs and moveable non-DTs to create workload networks.
3. Present two cluster-to-partition mapping strategies that ensure minimum inter-server data migrations and load-imbalance across partitions and servers.
4. Develop a scalable distributed data lookup technique that requires a maximum of two I/O roundtrips to locate a data tuple within the entire database.
5. Devise a set of quality metrics for the incremental repartitioning process defining different objectives.
6. Design and develop of a generic transaction generation model with comprehensive sensitivity analysis to demonstrate its capability for simulating wide variety of transactional systems for evaluating graph min-cut based incremental database repartitioning.
7. Develop a new definition for the impact of distributed transactions which can accurately measure the system-level impact of processing large number of DTs simultaneously and insensitive to workload variations.
8. Evaluate the performance of the proposed incremental repartitioning methods for both range and consistent-hash based initial data partitioning. This demonstrates the adaptability of our novel transaction generation model and ground truth about different repartitioning schemes over a common platform.

The remainder of this paper is organised as follows: we review the related works in Section 2; a high-level overview of the proposed framework is discussed in Section 3; Section 4 details the steps, formulations, modelling approaches, and design philosophies with necessary illustrations; Section 5 discusses the experimental results and remarks; and finally Section 6 concludes the paper.

2. Related works

Workload-aware load-balance with I/O overhead minimisation in distributed database systems was studied before with respect to parallel disk system [11] and for finding optimal data placement strategy in *shared-nothing* parallel databases [12]. Recent works primarily focus on OLTP workloads for scaling-out the Web applications to minimise the number of DTs. Graph min-cut based workload-aware data partitioning approach is first introduced by [5] for OLTP databases. The authors proposed 'Schism' which represents the transactional workload as a graph, and performs k-way replicated graph partitioning to minimise the effect of DTs. However, 'Schism' usually generates very large graphs, does not deal with dynamic workload changes, and the more general problem of incremental repartitioning. Transactional workloads are modelled as compressed hypergraph in [6,13] by hashing data tuple's primary key to reduce the overhead of k-way clustering. The authors propose 'SWORD', an incremental repartitioning technique which moves a fixed amount of data in a regular interval upon notifying workload changes, and by observing the increase in the percentage of DTs from a predefined threshold. However, this reactive approach only ensures local load-balance, and does not always guarantees reduction in DTs. Due to the selective swapping of the randomly compressed tuple sets and newly transformed DTs, the quality of min-cut clustering may lost, and gradually lead to global data distribution imbalance.

In [14], another automatic workload-aware database partitioning method is proposed along with an analytical model to estimate skew and coordination cost for DTs. It uses the same graph based workload representation of [5], and primarily focuses on optimal database design based on workload characteristics. However, it did not consider the necessity of incremental repartitioning as well. 'Elasca' is proposed in [15], where a multi-object workload-aware online optimiser is developed for optimal partition placement ensuring minimum data movement, however it does not support incremental repartitioning. In [2] the authors consider the problem of cost minimisation in big data processing systems and particularly investigate the relationship between physical data placement and routing. The scenario turns out to be an optimisation problem and mixed-integer linear programming (MILP) solver is used to find an optimal answer. By logically partitioning the physical data accesses, [16] propose physiological partitioning (PLP)—a transaction processing approach where physical access to database *pages* and *indexes* are continuously repartitioned and maintain load-balance based on data access patterns. However, applications that have less impact on the underlying storages can be penalised as mentioned by the authors, and understanding the directional flow and dependencies of data access for each transactions can be a costly operation. Similar approach is proposed in [17] as well which works at a higher granularity of assigning partitions of transactional access to threads.

Workload-aware data partitioning has been also studied with respect to social networks. In [18], a Social Partitioning and Replication middleware—(SPAR) is proposed that explores the social network graph from user interaction, and then performs joint partitioning and replication to ensure local data semantics for the users. Similarly, in [19], temporal activity hypergraphs are used to model user interactions in social network, and then min-cut clustering is used to minimise the impact of DTs with minimum load-imbalance. A distributed lookup method for transactional databases requiring special 'knowledge nodes' for coordination is proposed in [20], however it may perform incorrect routing due to inconsistent values. [5,13] also used a centralised data lookup scheme which is a clear bottleneck for high-scalability. In contrast, our proposed distributed lookup operation is based on the well known concept of *roaming* [21], and it always guarantees consistent results with a maximum of two lookups.

In overall, none of these techniques explore the incremental repartitioning problem, and the effect of physical data migrations in global load-balance. Moreover, the existing literature primarily focusing on reducing the percentage of DTs in a system for a particular observation. None of the previous studies propose a set of metrics which can clearly capture the actual impact of DTs on system wide resource consumption, incremental load-imbalance across the physical servers and logical partitions, and cost physical data migrations. Notably, [22] presents a detailed performance modelling foundation and framework for distributed and replicated databases, although it did not capture the necessity of incremental repartitioning at that time. Furthermore, apart from being evaluating the proposed architectures through heterogeneous benchmark environments, none of the existing works have considered modelling a theoretical framework to compare different repartitioning solutions within a common ground. We argue that there is a clear requirement to establish a theoretical ground for studying workload-aware incremental repartition schemes in a homogeneous manner that is independent of their benchmarking results.

3. System architecture

A high-level architecture of a *shared-nothing* distributed database cluster enabling workload-aware incremental repartitioning is shown in Fig. 1. We assume a set of *Transaction Coordinator* nodes residing in the application servers serving database queries generated by the higher level application processors. *Transaction Coordinators* are connected with a set of *shared-nothing Data Nodes* where the logical partitions are located. Each *Transaction Coordinator* contains a *Distributed Transaction Manager* that executes DTs or *XA* transactions [23] through the ODBC interface [24] exists between the *ODBC Manager* and the *ODBC Driver* in

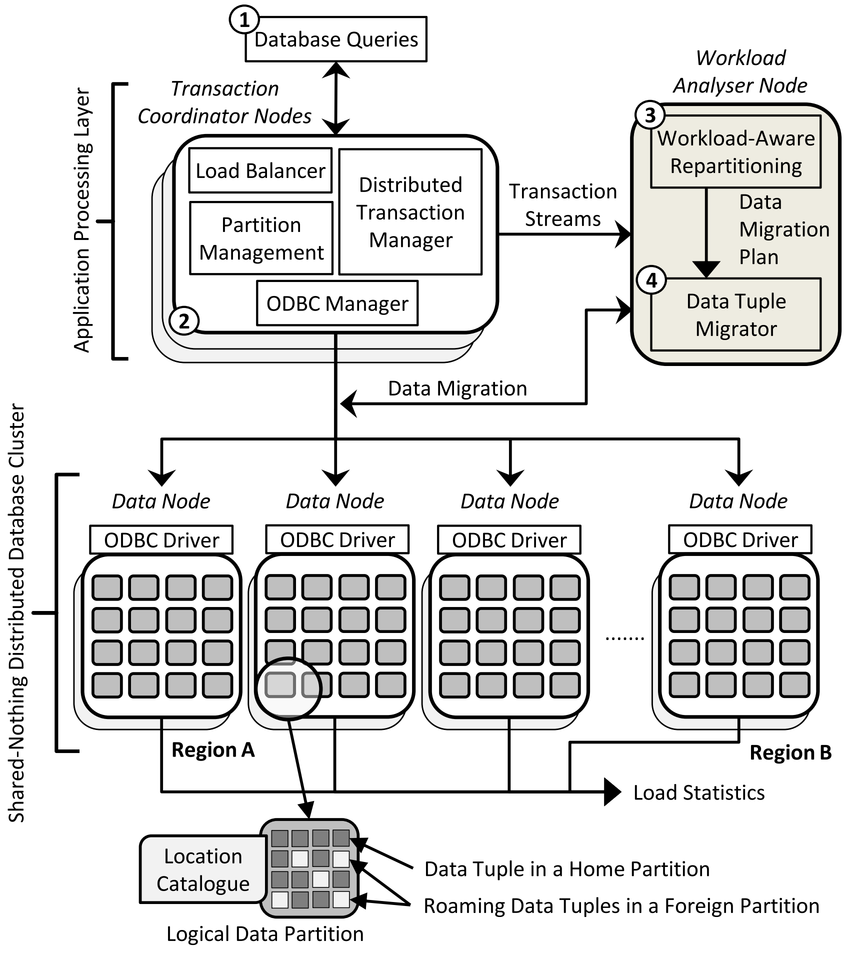


Fig. 1. High-level architecture of a shared-nothing distributed database cluster with workload-aware incremental repartitioning.

the *Data Node*. *Transaction Coordinators* also administrate partition management jobs (like split, merge, and migration) and balance incoming read/write workloads. Each logical partition in a *Data Node* contains a location catalogue that keeps track of the current location (i.e., current partition ids) of the data tuples as id–location pairs. Note that, individual *Data Nodes* can be synchronously replicated as master–slave within independent groups to ensure high availability which is a common deployment practice nowadays. Therefore, in this work we do not explicitly handle tuple level replication like [5].

Streams of transactional logs are continuously pulled by the *Workload Analyser Node*, and pre-process for analysis either in a time or workload sensitive window. *Workload Analyser Node* can also cache the most frequently appeared tuple location in a workload-specific catalogue which is kept updated upon inter-partition data migrations. Upon receiving *incremental repartitioning* command, the *Workload Analyser* starts processing the transactional logs to generate a suitable *data migration plan*. Fig. 2 details this procedure using blocks *a*–*g*. The input of the workload-aware incremental repartitioning component (in dotted rectangle) is the transactional logs while the output is a partition-level data migration plan. The overall process has four primary steps:

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 Data Node* cluster. Upon pulling the streams of transactional workloads in step 2, individual transactions are processed to extract the contained SQL statements at step 3a. For each SQL statement, the primary keys of individual tuples are extracted, and corresponding partition ids are retrieved from the embedded workload specific location catalogue in step 3b. In the classification process at step 3c, original DT and moveable non-DTs are identified along with their frequency counts in the current workload, and their associated costs of spanning multiple servers.

Workload representation and k-way Clustering. In step 3d, workload networks are generated from the extracted transactional logs gathered in the previous step using graph or hypergraph. Tuple-level compression can further reduce the size of workload network. Since transactional graphs cannot fully represent transactions with more than two tuples using pair-wise relationship, we cannot directly minimise the impact of DTs in the workload. However, graph representations are much simpler to produce, and it adopted wide ranges of application specific usages that also help us to understand its importance in creating workload networks. On the other hand, hypergraphs can exploit exact transactional relationships, thus the number of hyper edge cuts exactly matches the number of DTs. Yet, popular hypergraph clustering libraries are computationally slower than the graph clustering libraries, and produce less effective results [5].

In reality, with the increase in size and complexity, both of these representations are computation intensive in manipulation. Furthermore, compression techniques can confine an algorithm within a specified target, dramatic degradation in clustering quality and overall load-balance occur with a high compression ratio [6]. Finally, workload networks are clustered using k min-cut clustering employed by the graph and hypergraph clustering libraries in step 3e.

Cluster-to-partition mapping. At step 3f, a mapping matrix is created with the counts for tuples that are placed in the

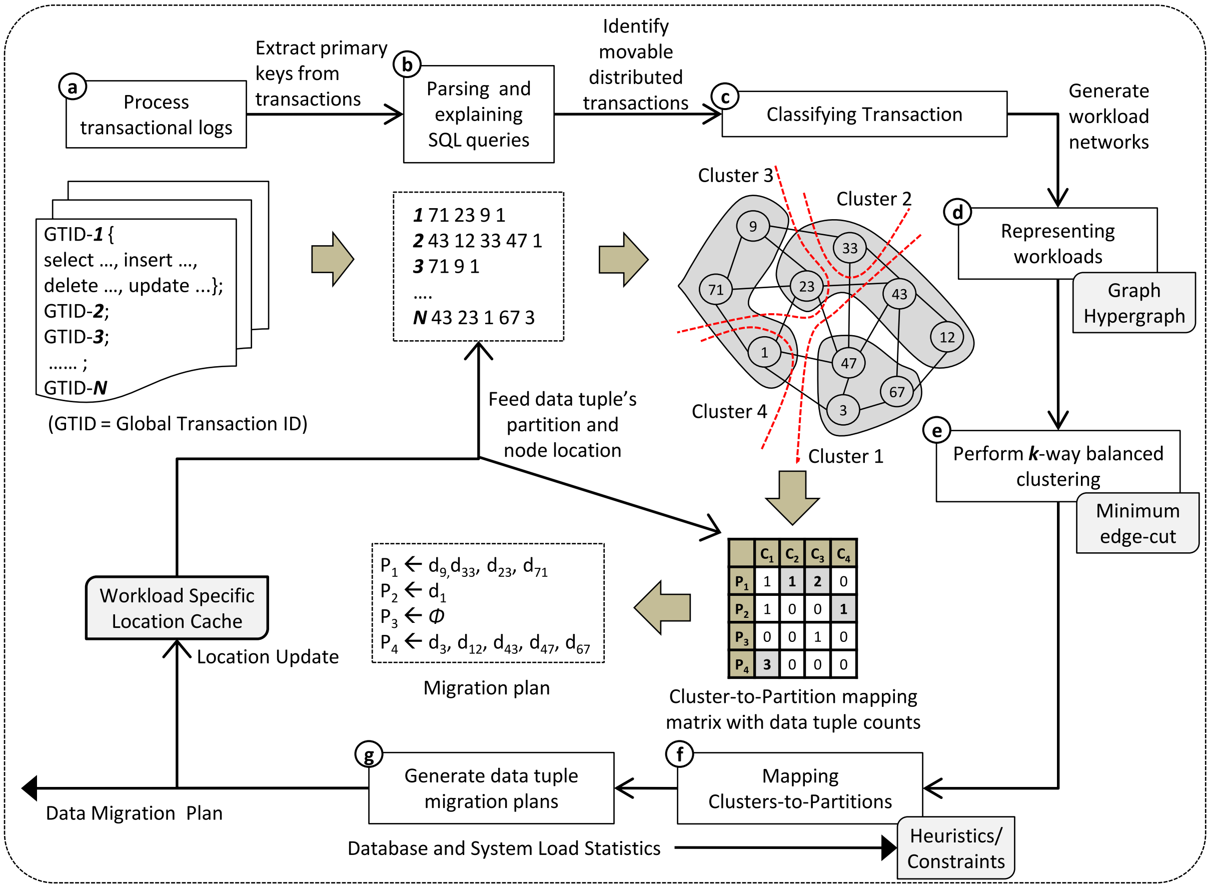

a
Process transactional logs
Extract primary keys from transactions
b
Parsing and explaining SQL queries
Identify movable distributed transactions
c
Classifying Transaction
Generate workload networks
d
Representing workloads
Graph Hypergraph
GTID-1 { select ..., insert ..., delete ..., update ...}; GTID-2; GTID-3; ...... ; GTID-N
(GTID = Global Transaction ID)
1 71 23 9 1
2 43 12 33 47 1
3 71 9 1
....
N 43 23 1 67 3
Cluster 3
Cluster 2
Cluster 4
Cluster 1
9
33
71
23
43
12
1
47
67
3
e
Perform k-way balanced clustering
Minimum edge-cut
Feed data tuple's partition and node location
Workload Specific Location Cache
Location Update
P1 ← d9, d33, d23, d71
P2 ← d1
P3 ← Φ
P4 ← d3, d12, d43, d47, d67
Migration plan
C1 C2 C3 C4
P1 1 1 2 0
P2 1 0 0 1
P3 0 0 1 0
P4 3 0 0 0
Cluster-to-Partition mapping matrix with data tuple counts
g
Generate data tuple migration plans
f
Mapping Clusters-to-Partitions
Data Migration Plan
Database and System Load Statistics
Heuristics/ Constraints

Fig. 2. An overview of the workload-aware incremental repartitioning framework using numbered notations. Steps 3a–g represent the flow of workload analysis, representation, clustering, and repartitioning decision generation.

newly created cluster and originated from the same partition as the matrix element. The produced clusters from the min-cut clustering are then mapped to the existing set of logical partitions by following three distinct strategies. At first, we employ uniform random tuple distribution for mapping clusters to database partitions which naturally balances the distribution of tuples over the partitions. However, there is no proactive consideration in this random strategy for minimising data migrations. The second strategy employs a straight forward but optimal approach. It maps a cluster to a respective partition which originally contains 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 diverges uniform tuple distribution. Again, incremental repartitioning can create server hot-spot as similar transactions from new workload batches 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 3g, based on different mapping strategies and applied heuristics a data migration plan is generated, and then forwarded to the data tuple migrator module in step 5.

Distributed location update and routing. The analyser node keeps a workload specific location catalogue for the most frequently accessed tuples, and updates the associated locations at each repartitioning cycle in step 4. The analyser also directly invokes the corresponding data nodes to perform data migrations in step 6 without interrupting the ongoing transactional services. Until a tuple fully migrates to a new partition, its existing partition serves all the query requests. Distributed databases using range partitioning require keeping a central lookup table for the clients to retrieve tuples. Hash partitioning requires the client to use a fixed hash function to lookup the required tuples in the specified server. Consistent hash partitioning [25] employs distributed lookup mechanism using distributed hash table. However, none of these partitioning schemes provide scalable data lookup mechanisms for successive data redistribution.

To solve this problem, we use the well established concept of *roaming* from wireless telecommunications and computer data networks. The problem of location independent routing is already solved in IPv6 using Mobile IP [26], and in GSM networks using roaming mobile stations [21]. In a similar way, the attached location catalogue within each data partition keeps track of the *roaming* tuples and their corresponding *foreign* partitions. A maximum of two lookups are required to find a tuple without client-side caching. With proper caching enabled, this lookup cost can be even amortised to one for most of the cases with high cash hit.

4. Workload-aware incremental repartitioning

4.1. Problem formulation

Let $S = \{S_1, \dots, S_n\}$ be the set of n *shared-nothing* physical database servers where each $S_i = \{\mathcal{P}_{i,1}, \dots, \mathcal{P}_{i,m}\}$ denotes the set of m logical partitions reside in S_i. Let $\mathcal{D}_{\mathcal{P}_{i,j}} = \{\delta_{i,j,1}, \dots, \delta_{i,j,|P_{i,j}|}\}$ denotes the set of data tuples reside in $\mathcal{P}_{i,j}$. Hence, we get the total number of data tuples reside in S_i as $|\mathcal{D}_{S_i}| = \bigcup_{\forall j} |\mathcal{D}_{\mathcal{P}_{i,j}}|$. Finally,

$|\mathcal{D}_S| = \bigcup_{\forall i} |\mathcal{D}_{S_i}|$ denotes the total number of data tuples in the entire partitioned database.

Let $\mathcal{W}$ be the observation window consisting of $|\mathcal{W}|$ number of preceding transactions at the time of repartitioning. Let the transactional workload in $\mathcal{W}$ be represented by $T = \{\tau_1, \ldots, \tau_{|T|}\}$ where τ_i's are the $|T|$ unique transactions observed in $\mathcal{W}$. Note that, repartitioning decision is made considering only the unique transactions and hence the term 'unique' will be dropped in the rest of the paper. Furthermore, the set of distributed and non-distributed transactions 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 $T_d = \{\tau_{d_1}, \ldots, \tau_{d_{|T_d|}}\}$ and $T_{\hat{d}} = \{\tau_{\hat{d}_1}, \ldots, \tau_{\hat{d}_{|T_{\hat{d}}|}}\}$. Any distributed or non-distributed transaction τ_{d_i} or $\tau_{\hat{d}_i}$ can appear (i.e., repeat) multiple times within $\mathcal{W}$ hence its frequency can be represented by either $f(\tau_{d_i})$ or $f(\tau_{\hat{d}_i})$. As any τ_{d_i} can span multiple servers, we define the cost of spanning as $s(\tau_{d_i})$. We consider the cost of spanning multiple partitions by a non-distributed transaction negligible in terms I/O overhead and define $s(\tau_{\hat{d}_i}) = 1$. Lets now define the problem of incremental repartitioning as:

Problem definition: For a given observation window $\mathcal{W}$, $\mathcal{S}$ homogeneous servers containing a total of $\mathcal{P}$ logical partitions, and a maximum allowed imbalance ratio ϵ, find an incremental repartitioning solution $\mathcal{X}_i$ from the output of a k-way balanced clustering ζ which minimises the impact of DTs in $\mathcal{W}$ and imbalance in $\mathcal{D}_S$ by performing minimum inter-server data migrations.

4.2. Workload modelling

We model the workload networks using three distinct representations. Firstly, *Graph Representation (GR)* produces fine-grain workload network although it is unable to fully capture the actual transactional relationship between different tuples. Yet, graph *min-cut* process can still generate high quality k-way clustering and minimises the impact of DTs, unless the overall graph size increases with workload variability, and adequate level of sampling is performed [5]. Secondly, *Hypergraph Representation (HR)* generates most accurate, and exact workload networks thus also able to produce balanced clusters with min-cut hypergraph clustering. Moreover, from our empirical studies we found that, k-way min-cut balanced hypergraph clustering produces more consistent results in terms of achieving the repartitioning goals, and is also mentioned in [19]. Finally, *Compressed Hypergraph Representation (CHR)* produces coarse-grain workload networks depending on the compress level. With lower level of compression, less coarse networks are generated and k-way clustering performs better. However, as shown in [6], as the level of compression increases the quality of the clustering process degrades dramatically. We formally define the individual representations as in below:

4.2.1. Graph representation

A graph $\mathcal{G} = (\mathcal{V}, E_g)$ represents $\mathcal{W}$ where each edge $e_g \in E_g$ links a pair of tuples (v_x, v_y) from $\mathcal{V} = \{v_1, \ldots, v_{|\mathcal{V}|}\} \subset \mathcal{D}_S$ for a transaction τ_i where $v_i = \exists a \exists b \exists c\, \delta_{a,b,c}$. Individual tuples from (v_x, v_y) connects to their respective set of adjacent tuples $\mathcal{A}_{v_x}$ and $\mathcal{A}_{v_y}$ originated from the same τ_i. Any edge within τ_i has a weight representing the frequency of τ_i in $\mathcal{W}$ which co-access the pair (v_x, v_y), while vertex weight represents the tuple's size (in volume).

4.2.2. Hypergraph representation

A hypergraph, $\mathcal{H} = (\mathcal{V}, E_h)$ represents $\mathcal{W}$ where a hyperedge $e_h \in E_h$ characterises a transaction τ_i and overlays its contained set of tuples $\mathcal{V}_{\tau_i} \subset \mathcal{V}$. A hyperedge representing τ_i is associated with a weight denoting the frequency of e_h within $\mathcal{W}$ and its vertices' weight represent data tuples' size (in volume).

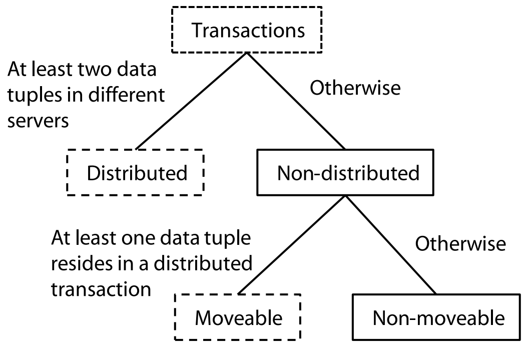

Transactions
At least two data tuples in different servers
Otherwise
Distributed
Non-distributed
At least one data tuple resides in a distributed transaction
Otherwise
Moveable
Non-moveable

Fig. 3. Transaction classification identifying DTs and moveable non-DTs.

4.2.3. Compressed hypergraph representation

A hypergraph, $\mathcal{H} = (\mathcal{V}, E_h)$ can be compressed by collapsing the vertices to a set of virtual vertices $\mathcal{V}'$ using a simple hash function on the primary keys [6]. A compressed hypergraph $\mathcal{H}_c = (\mathcal{V}', E'_h)$ represents $\mathcal{W}$ where each virtual hyperedge $e'_h \in E'_h$ constitutes the set of virtual vertices $v'_{e_h} \subset \mathcal{V}'$ where the original vertices of e_h are mapped into and $|v'_{e_h}| \geq 2$. Virtual vertex weight represents the combined data volume sizes of the corresponding compressed tuples. And hyperedge weight represents the frequency of transactions which access the corresponding virtual vertices. C_l denotes the compression level as $|\mathcal{V}|/|\mathcal{V}'|$ and equals to 1 for no compression while to $|\mathcal{V}|$ for full compression.

4.3. Proactive transaction classification

In constructing the classification technique, we argue that there always exists a group of tuples which are retrieved while processing the DTs, and also participated in the execution of non-distributed but frequently occurred transactions. These particular groups of tuples when move into different database servers due to the database repartitioning process can turn the previously non-DTs into newly distributed ones. We use this intuitive to classify the workload transactions into three different categories—distributed, non-distributed moveable and non-distributed non-moveable as shown in Fig. 3. As an example, transactions τ_1, τ_2, and τ_5 from the sample workload of Table 2 are identified as *distributed*, whereas τ_3 and τ_4 are labelled as *moveable non-distributed*. Finally, τ_6 and τ_7 are discarded as purely *non-distributed* transactions.

Clearly, a number of non-distributed moveable transactions will be remain protected within k-way clustering as the *min-cut* clustering always tries to preserve as much as transactional edges it could. As the tuples in these moveable transactions did not participate into any DTs, they are residing in isolation within the workload network. Thus, they are highly likely to be preserved together in the same cluster after k-way clustering. If we added the DTs τ_1, τ_2, and τ_5 in the workload sub-graphs, then at the next incremental repartitioning phase τ_3 and τ_4 would have been appeared as DT. Since, tuple with id 9, which by this time would have been already moved to another partition located in a different physical server, would cause its associated transactions to become distributed. There exists a clear trade-off between the increase of size of the workload networks and achieved benefits. At one end, the smaller is the workload network, it will less computationally costly to process with respect to time and I/O. On the other hand, if we include all the workload tuples in the representations, it may reduce the impact of DT better than in a particular repartitioning cycle, but with the price of unwanted data migrations to create new DTs. By aggressively classifying the *non-distributed moveable* transactions, the quality of the overall repartitioning process increases as the impact of DTs decreases comparing to a static partitioning strategy as shown later in our experimental results.

Table 1
Sample database—physical and logical layout.

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
Sample transactional workload for illustration.

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

4.4. *k*-way balanced clustering of workload

Given $\mathcal{G}$ and a maximum allowed imbalance ratio ϵ, we can define the problem as find the k-way clustering $\zeta_{\mathcal{G}} = \{V_1, \ldots, V_k\}$ that minimises transactional *edge cut* with the *balance* constraint bounds by $(1+\epsilon)$. Similarly, the k-way constrained and balanced clustering of $\mathcal{H}$ is $\zeta_{\mathcal{H}} = \{V_1, \ldots, V_k\}$ such that minimum number of hyper edges are cut having the imbalance ratio ϵ. Analogously, the k-way balanced clustering of $\mathcal{H}_c$ is $\zeta_{\mathcal{H}_c} = \{V'_1, \ldots, V'_k\}$ with an imbalance ratio ϵ aiming at minimum virtual hyperedge cuts. Note that, we denote k as the total number of logical partitions instead of the number of physical servers. From our empirical experiments we find that executing the k-way clustering processing with k as the number of partitions provide finer granularity in balancing the distribution of data volume over the set of physical servers.

The k-way balanced clustering generates clusters of similar size with respect to the number of tuples given a *balance* constraint which is defined as $k(\max(\omega_{V_i})/\omega_V)$, and tells whether the clusters are equally-weighted or not. Here, (w_{V_i}) is the sum of the weights of the vertices in V_i. The partitions are said to be balanced if the *balance* measure is equals to or close to 1 otherwise imbalanced if greater than 1.

4.5. Cluster-to-partition mapping strategies

In the following, we use illustrative examples using a simple database construction with 20 data tuples distributed using hash-partitioning over 4 logical partitions and 2 physical servers as shown in Table 1. A sample workload batch with 7 transactions and corresponding data tuples are also shown in Table 2. Finally, a detail illustration on how the cluster-to-partition mapping strategies work with different workload representations are shown in Fig. 4. In these figures the workload networks as *GR*, *HR*, and *CHR* (with $C_l = 2$) for the transactions listed in Table 2. Three distinctive cluster-to-partition mapping strategies (in matrix format) are also shown beneath their respective workload network representations. The rows and columns of the matrices represent partition and cluster id respectively. 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 matrix with the counts in *bold face* represents the resulting decision block 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 face* numbers in the layout tables at bottom denote most balanced distribution and least count for data migrations.

4.5.1. Random mapping (RM)

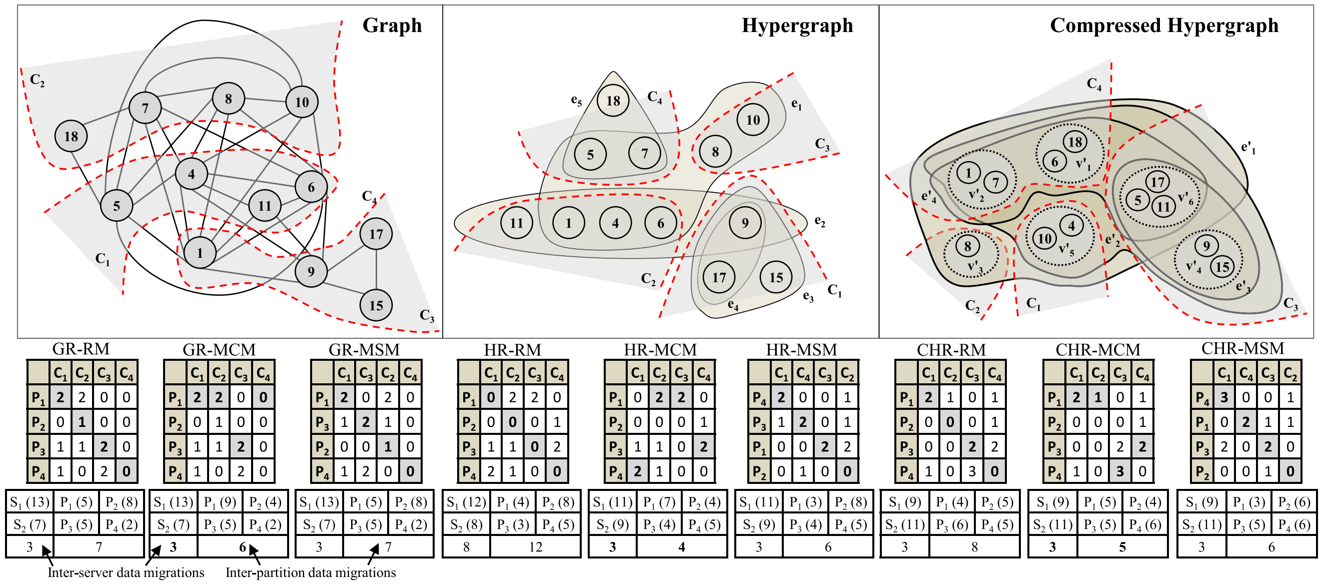
Naturally, the best way to achieve load-balance in any granularity is to assign the clusters randomly. Clustering tools like Metis and hMetis randomly generates the cluster ids, and do not have any knowledge about how the data tuples are originally distributed within the servers or partitions. As a straightforward 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 not necessarily guarantees minimum inter-server data migrations. As shown in Fig. 4, the mapping matrices labelled with *RM* and database layouts with *GR-RM*, *HR-RM* and *CHR-RM* are the representatives of this class.

4.5.2. Maximum column mapping (MCM)

We aim at minimising the physical data migration within the repartitioning process using this strategy. 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 represented partition id of that maximum count. If multiple maximum counts are found then we choose the one directing the partition containing lowest number of data tuples. Thus, multiple clusters can be assigned to a single partition. As maximum numbers of tuples are originated from this designated partition, they do not tend to leave from their home partitions which reduce the overall inter-server physical data migrations. For OLTP workloads with skewed tuple distributions and dynamic data popularity, the impact of DTs can rapidly decrease from this greedy heuristic as tuples from multiple clusters may map to a single partition in the same physical server. However, this may lead to data volume imbalance across the partitions and servers besides our selection preference of the partition id in the mapping matrix. Mapping matrices labelled as *MCM* with corresponding database layouts of *GR-MCM*, *HR-MCM*, and *CHR-MCM* represent this mapping strategy in Fig. 4.

4.5.3. Maximum submatrix mapping (MSM)

To both minimise load-imbalance and data migrations, we fork lift the natural advantages of the previous strategies and combine them together. 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 search the remaining *submatrices* for element with maximum tuple counts. Finally, all the diagonal positions of the matrix are filled up with elements having maximum tuple counts. Now, mapping the respective clusters one-to-one to the corresponding partitions results both minimum data migrations and distribution load-balance. Note that, multiple maximum tuple counts can be found in different matrix positions, and the first such encountered element is chosen for simplicity. The *MSM* strategy works similarly to the *MCM* strategy as it prioritises the maximum tuple counts within the sub-matrices, and map 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 Fig. 4, mapping matrices labelled as *MSM*, and representative database layouts *GR-MSM*, *HR-MSM*, and *CHR-MSM* depict this mapping strategy.



GR-RM

	C_1	C_2	C_3	C_4
P_1	2	2	0	0
P_2	0	1	0	0
P_3	1	1	2	0
P_4	1	0	2	0

S_1 (13)	P_1 (5)	P_2 (8)
S_2 (7)	P_3 (5)	P_4 (2)
3	7	

GR-MCM

	C_1	C_2	C_3	C_4
P_1	2	2	0	0
P_2	0	1	0	0
P_3	1	1	2	0
P_4	1	0	2	0

S_1 (13)	P_1 (9)	P_2 (4)
S_2 (7)	P_3 (5)	P_4 (2)
3	**6**	

GR-MSM

	C_1	C_3	C_2	C_4
P_1	2	0	2	0
P_3	1	2	1	0
P_2	0	0	1	0
P_4	1	2	0	0

S_1 (13)	P_1 (5)	P_2 (8)
S_2 (7)	P_3 (5)	P_4 (2)
3	7	

HR-RM

	C_1	C_2	C_3	C_4
P_1	0	2	2	0
P_2	0	0	0	1
P_3	1	1	0	2
P_4	2	1	0	0

S_1 (12)	P_1 (4)	P_2 (8)
S_2 (8)	P_3 (3)	P_4 (5)
8	12	

HR-MCM

	C_1	C_2	C_3	C_4
P_1	0	2	2	0
P_2	0	0	0	1
P_3	1	1	0	2
P_4	2	1	0	0

S_1 (11)	P_1 (7)	P_2 (4)
S_2 (9)	P_3 (4)	P_4 (5)
3	**4**	

HR-MSM

	C_1	C_4	C_3	C_2
P_4	2	0	0	1
P_3	1	2	0	1
P_1	0	0	2	2
P_2	0	1	0	0

S_1 (11)	P_1 (3)	P_2 (8)
S_2 (9)	P_3 (4)	P_4 (5)
3	6	

CHR-RM

	C_1	C_2	C_3	C_4
P_1	2	1	0	1
P_2	0	0	0	1
P_3	0	0	2	2
P_4	1	0	3	0

S_1 (9)	P_1 (4)	P_2 (5)
S_2 (11)	P_3 (6)	P_4 (5)
3	8	

CHR-MCM

	C_1	C_2	C_3	C_4
P_1	2	1	0	1
P_2	0	0	0	1
P_3	0	0	2	2
P_4	1	0	3	0

S_1 (9)	P_1 (5)	P_2 (4)
S_2 (11)	P_3 (5)	P_4 (6)
3	**5**	

CHR-MSM

	C_1	C_4	C_3	C_2
P_4	3	0	0	1
P_1	0	2	1	1
P_3	2	0	2	0
P_2	0	0	1	0

S_1 (9)	P_1 (3)	P_2 (6)
S_2 (11)	P_3 (5)	P_4 (6)
3	6	

Fig. 4. *k*-way min-cut clustering of transactional workload networks followed by 3 cluster-to-partition mapping strategies.

4.6. Distributed data lookup

As mentioned in Sections 1 and 3, any centralised lookup mechanism is always at risk to be the bottleneck in achieving high-availability and scalability requirements. We take a sophisticated approach to distribute the data tuple lookup process into individual database partition level. Thus, data migration operations are totally transparent to distributed transaction processing and coordination. By maintaining a key-value list of *roaming* and *foreign* data id with their corresponding partition id, individual partitions can answer the lookup queries. Tuples are assigned permanent *home* partition id for its lifetime when the database is initially partitioned using range, hash, or consistent hash [25]. *Home* partition id only changes while a partition splits or merges and these operations are overseen by the *transaction coordinators* as shown in Fig. 1, thus transparent to the lookup process. As the tuple locations are managed by their *home* partitions, data inconsistency are strictly prevented. Unless a tuple is fully migrated to another partition, and its *roaming* location is written in the catalogue, the old partition continue serving transactional processing.

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 will treat this migrated tuple as a *foreign* and updates its lookup table accordingly whereas the original *home* partition will mark this tuple as *roaming* in its lookup table and update its current location with the *roaming* partition's id. A lookup process always query the tuple's *home* partition to retrieve it. If the tuple is not initially found in its original location, the lookup table entry thus immediately informs the most recent location of the tuple and redirect the search towards the *roaming* partition. Thus, a maximum of two lookup operations can be required to find a tuple within the entire database.

Note that, the cost of physical data migration may increase while using such distributed lookup process. With a high probability individual data migrations in the incremental repartition process may involve running location update process up to three physical servers serving the *home* partition and two *roaming* partitions—current and target partitions. At present, we are investigating the implication of this cost, and how to include this in the formulation of the quality measures.

4.7. Transaction generation model

In order to evaluate the performance of the proposed graph min-cut based repartitioning algorithms, we need to be careful to avoid any undue influence of the external graph partitioning library. The easiest way to achieve this is to carefully design 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 graphs 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. 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 purpose. Let $\eta = \mathcal{R}p$ be the average generation rate of new transaction in the current observation window. Obviously $\mathcal{U}$ is bounded as follows:

$$\eta \leq \mathcal{U} \leq \mathcal{R}. \tag{1}$$

Implementation of this transaction model is quite straightforward. The only challenge remains is to set a limit on how far back the generator should look into 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, we may now restrict $\mathcal{U}'$ within the following lower and upper bounds

$$\mathcal{U} - \eta \leq \mathcal{U}' \leq \mathcal{U}. \tag{2}$$

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

$$\mathcal{U}' = \mathcal{U}\left(1 - \left(\frac{\eta}{\mathcal{U}}\right)^q\right) \tag{3}$$

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

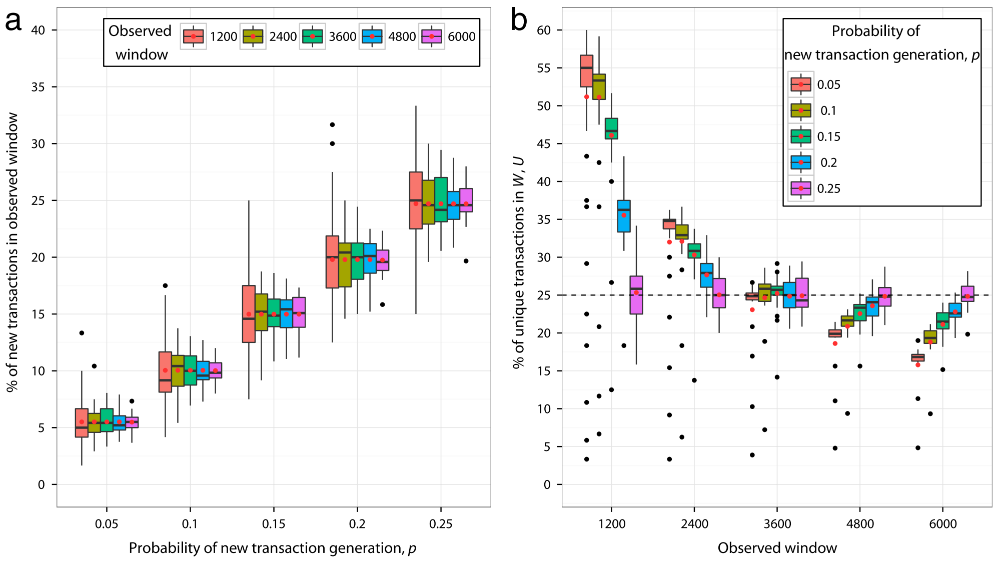


Fig. 5. Transactional workload generations validating (a) the consistency of generating a target percentage of new transactions for different observation windows for a given probability of new transaction generation, and (b) consistency of generating a target percentage of unique transactions for different probabilities of new transaction generation against different observation windows.

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 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 $\mathcal{W}$.

4.7.1. *Sensitivity analysis*

We perform a detail sensitivity analysis of the above described transaction generation model for different combinations of $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p\rangle$ by varying p. We choose an average transaction generation rate $\mathcal{R}$ of 1 transaction per second (tps) and a target of achieving 25% unique transactions in a given $\mathcal{W}$ of 3600s (1 h) containing 3600 preceding transactions to generate transactional workloads for a total of 24 h. The value of p is varied from 0.05 to 0.25 by 0.05. Fig. 5 shows – (a) the generation of new transactions against different values of p observed over 5 distinct observation windows containing – 1200, 2400, 3600, 4800, and 6000 transactions; and (b) the generation of unique transactions against these observation windows grouped by p values. From the results of Fig. 5(a), the consistency of the proposed transaction generation model in generating new transactions for any given p within different observation windows are noticed. Fig. 5(b), on the other hand, shows the consistency of generating a target of 25% unique transactions within every 3600 preceding transactions generated in $\mathcal{W}$.

As mentioned earlier, following (2) and (3), if we vary the observation window from the given $|\mathcal{W}|$ the target $\mathcal{U}$ will be either underachieved or overachieved. These phenomenons are observed from the grouped bar plots for the observation windows of 1200 and 2400 where the target $\mathcal{U}$ was overachieved while it was underachieved for window sizes 4800 and 6000 except where $p = 0.25$. As per our experimental setting $p = 0.25$ determines that for a given $\mathcal{R}$ of 1 tps 25% new transactions will be generated within any $|\mathcal{W}|$. At the same time, we also set an initial target $\mathcal{U}$ for generating 25% unique transactions considering $|\mathcal{W}|$ given $R = 1$—a fixed amount of 3600 preceding transactions. The results of Fig. 5(b) thus holds the relationship of (2) which restricts the repetitions of newly generated transactions. Furthermore, if we decrease the window size to 1200 and 2400 then with the decrement of p the target number of unique transactions are over achieved. In contrast, if we increase the window size to 4800 and 6000 from 3600, the target number of unique transactions are under achieved with the increment of p. When the value of p matches with the proportion of unique transaction generation set as $\mathcal{U}$, no transactional repetitions is took place according to (3). Table 3 lists the statistics of transactional repetitions for the given $|\mathcal{W}|$ of 3600 transactions varying p. From the statistics, at $p = 0.25$ inter-repetition interval reaches to 3600 which leaves no unique transactions for repetition thus all the generated transactions are new. In overall, by setting appropriate parameter values of the proposed transaction generation model one can easily generate transactional workload with desired repetition, uniqueness, and new transaction generation properties.

4.8. *Quality metrics for incremental repartitioning*

In evaluating the performance of the incremental repartitioning, previous works [5,13] only measure the percentage of reduction in DTs. However, this single measure fails to imply any meaning conclusion about how the impact of distributed transaction is minimised. Further, there are no measures for overall load-balance and data migrations. We propose three independent metrics to measure the successive repartitioning quality achieving three distinct objectives—(1) minimise the impact of DTs; (2) minimise load-imbalance; and (3) minimise the number of physical data migrations. The first metric measures the impact associating the frequency of DTs and their related spanning cost which is directly related to system resource consumption. The second metric measures the tuple-level load distribution over the set of servers using *coefficient of variation* which effectively shows the dispersion of data load over successive period of observations. The third metric measures the mean inter-server data migrations for successive repartitioning processes. By combining all three aforementioned mentioned metrics, a composite metric is also proposed which will able tell us the required combination of different workload representations and cluster-to-partition mapping strategies for a particular incremental repartitioning cycle to achieve a certain objective. The detail modelling follows in below:

4.8.1. *The impact of distributed transaction*

Considering the formal definitions provided in Section 4.1, for any given τ_{d_i} we multiply the spanning cost $s(\tau_i)$ with its frequency

Table 3
Transactional repetition statistics for different values of p given $\langle \mathcal{R}, p, \mathcal{W}, \mathcal{U} \rangle = \langle 1, 3600, 0.25 \rangle$.

p	0.05	0.10	0.15	0.20	0.25
Number of unique transactions used for repetition, $\mathcal{U}'$	864	756	576	324	0
Mean inter-repetition interval	768.1072	645.5693	474.8184	261.9204	3600
Mean repetition frequency	4.6868	5.5764	7.5818	13.7446	1

within $\mathcal{W}$, $f(\tau_{d_i})$ to get the total cost of distributed transaction for τ_{d_i}. Here, $s(\tau_{d_i}) = S_{\tau_{d_i}} = \{\forall v \in \tau : a \mid \exists a \exists b \exists c\ \delta_{a,b,c} = v\}$ which denotes the number of physical servers involved in processing τ_{d_i}, whereas, $s(\tau_{\hat{d}_i}) = 1$ for any non-distributed transaction. Note that, in reality this cost represents the overhead of I/O over the network while processing the DTs. Eq. (4) defines the spanning cost of T_d within $\mathcal{W}$ for all $\tau_{d_i} \in T_d$

$$c(T_d) = \sum_{\forall \tau_{d_i} \in T_d} f(\tau_{d_i}) s(\tau_{d_i}). \tag{4}$$

Similarly, (5) denotes $c(T_{\hat{d}})$ for all $\tau_{\hat{d}_i} \in T_{\hat{d}}$

$$c(T_{\hat{d}}) = \sum_{\forall \tau_{\hat{d}_i} \in T_{\hat{d}}} f(\tau_{\hat{d}_i}). \tag{5}$$

Finally, the impact of distributed transaction is defined as:

$$I_d(\mathcal{W}) = \frac{c(T_d)}{c(T_d) + c(T_{\hat{d}})}. \tag{6}$$

According to the definition in (6), the impact of distributed transaction is estimated in real domain within the range [0, 1]; the lower and upper limit are reached when there is no distributed transaction ($T_d = \phi$) and no non-distributed transaction ($T_{\hat{d}} = \phi$), respectively. This impact metric, however, suffers from the following shortcomings:

1. It is insensitive to the server spanning cost of the distributed transactions when they significantly outnumber non-distributed transactions, 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 $[0.95, 1)$.
2. 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, the impact metric in two reasonably overlapped windows with the same set of unique transactions may vary significantly due to frequency differences.

It is, therefore, reasonable for this paper to deviate from the definition of impact we presented in its preliminary version [27]. 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 kth occurrence of the transaction τ in the system from the start of the system. Its observed period $r_\tau(k)$ and expected period of recurrence $\tilde{r}_\tau(k)$ is updated as follows:

$$r_\tau(k) = t_\tau(k) - t_\tau(k-1) \tag{7}$$

for $k > 1$ and

$$\tilde{r}(k) = \begin{cases} \mathcal{U}' & k = 1 \\ \alpha r(k) + (1-\alpha)\tilde{r}(k-1) & k > 1 \end{cases} \tag{8}$$

where the coefficient α represents the degree of weighting decrease, a constant smoothing factor 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) from the target number of unique transactions $\mathcal{U}$ and the average number of new transactions added in the observation window $\mathcal{W}$, under uniform recurrence assumption.

Now, let $T_u = \{\tau_{u_1}, \ldots, \tau_{u_{|T_u|}}\}$ be the set of unique transactions in $\mathcal{W}$. We may now estimate the impact of distributed transactions in $\mathcal{W}$ as

$$I_d(\mathcal{W}) = \frac{\sum_{\forall j} s(\tau_{u_j}) f(\tau_{u_j})}{|S| \sum_{\forall j} f(\tau_{u_j})} = \frac{\sum_{\forall j} \Big(s(\tau_{u_j}) / \tilde{r}_{\tau_{u_j}}(k_j) \Big)}{\sum_{\forall j} \Big(|S| / \tilde{r}_{\tau_{u_j}} \Big)} \tag{9}$$

where k_j denotes the number of occurrences of unique transaction τ_{u_j} from the start of the system. Note that, according to (8), $I_d(\mathcal{W})$ is bounded in the range $[1/|S|, 1]$ and hence it is also an indicator of system resource consumption due to a certain transactional workload.

4.8.2. 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 we compute the standard deviation of data volume $\sigma_{|\mathcal{D}_S|}$ for all the physical servers, 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 $\mu_{|\mathcal{D}_S|}$ for all S under deployment, and independent of the unit of measurement. C_v can tell the variability of tuple distribution within the servers in relation to the mean data volume $\mu_{|\mathcal{D}_S|}$. Eq. (10) determines the C_v of the server-level load-balance of the entire database cluster independent of any observation instance.

$$L_b = \frac{\sigma_{|\mathcal{D}_S|}}{\mu_{|\mathcal{D}_S|}} \tag{10}$$

where

$$\mu_{|\mathcal{D}_S|} = \frac{1}{n} \sum_{i=1}^{n} |\mathcal{D}_{S_i}|$$

$$\sigma_{|\mathcal{D}_S|} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} \Big(|\mathcal{D}_{S_i}| - \mu_{|\mathcal{D}_S|} \Big)^2}.$$

4.8.3. Inter-server data migrations

At any $\mathcal{W}$, the total of inter-server data migrations within a repartitioning process can be normalised by dividing the total number inter-server data migrations by the mean data volume $\mu_{|\mathcal{D}_S|}$. As shown in (11), D_m is the inter-server data migration metric with respect to an observed $\mathcal{W}$ during a particular repartitioning process.

$$D_m = \frac{M_v}{\mu_{|\mathcal{D}_S|}} \tag{11}$$

where M_v is the total number of migrations during the observed $\mathcal{W}$.

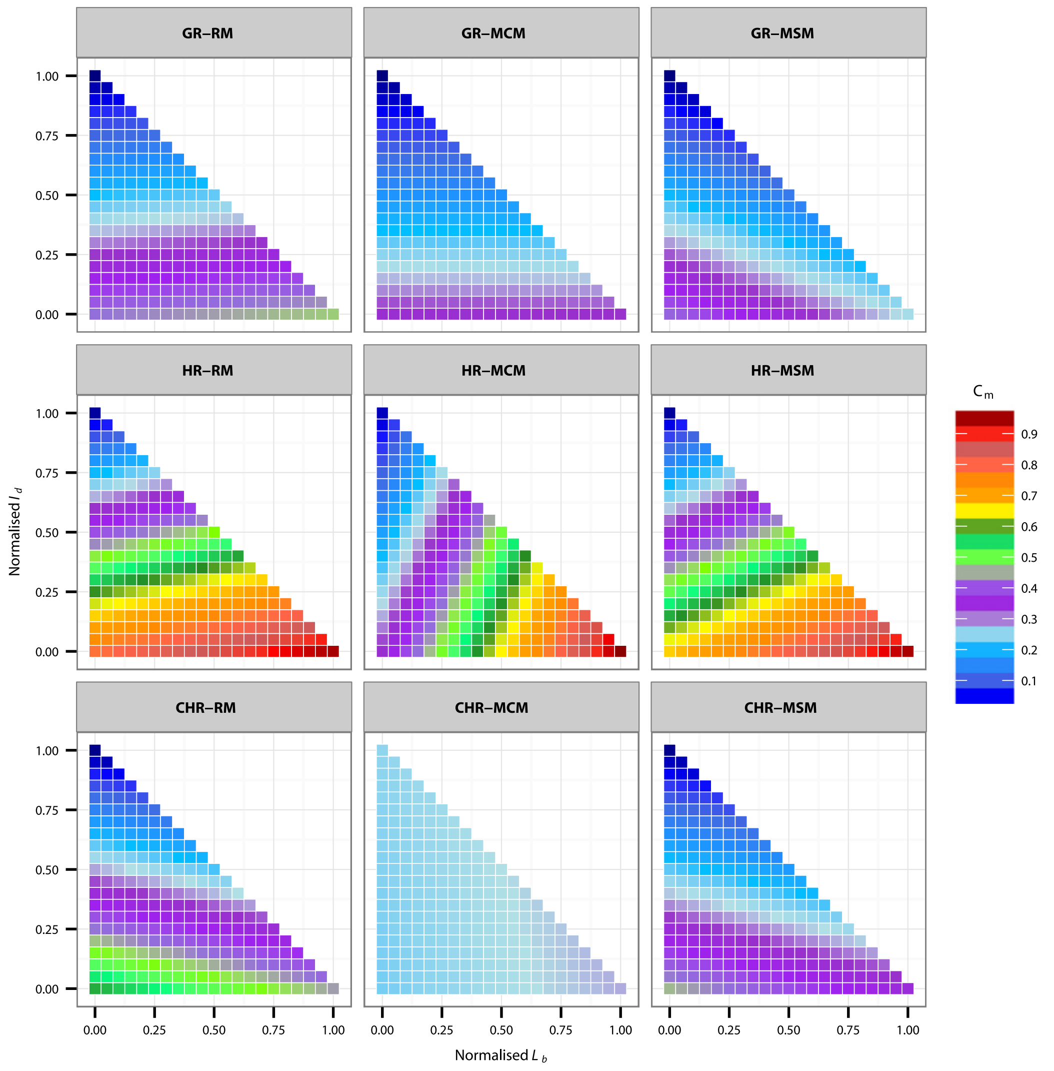


Fig. 6. Combined effect of I_d, L_b, and D_m through composite metric C_m. Note that, lower values of C_m indicate better solutions. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

4.8.4. Composite metric

Let C_m be the composite metric with weight factors ω_{I_d}, ω_{L_b}, and ω_{D_m} respectively for the objective measures I_d, L_b, and D_m where $\omega_{I_d} + \omega_{L_b} + \omega_{D_m} = 1$ providing two degrees of freedom to choose between different repartitioning goals. Given the application and system requirements, system administrators can set specific goal towards achieving certain quality objectives—*minimise* I_d, L_b, or D_m for the incremental repartitioning process. Based on different weight distributions, it is thus possible to find a repartitioning sweet-spot preferring particular choices of workload network representation and cluster-to-partition mapping strategy. By fine tuning the combinations in weight distribution one can instantly tackle unpredictable situations by tweaking the direction of incremental repartitioning process to maintain acceptable level of transactional services. We define the C_m as follows:

$$C_m = \omega_{I_d} I_d + \omega_{L_b} L_b + \omega_{D_m} D_m. \tag{12}$$

Based on our preliminary paper [27], the combined effect of I_d, L_b, and D_m is examined using (12) with different combinations of weight factors such that $\omega_{I_d} + \omega_{L_b} + \omega_{D_m} = 1$. Fig. 6 shows the resulting measure of C_m in a 2-d perspective plot using coloured scale where L_b and I_d are plotted in the X-axis and Y-axis respectively. The locations presenting the values of D_m can be determined by calculating $1 - (\omega_{I_d} + \omega_{L_b})$ in the individual subplots. We can set specific preferences to prioritise one particular repartitioning *quality measure* over other. Individual extremes of I_d, L_b, and D_m can be found at (1, 0), (0, 1), and (0, 0) locations. By following the colour codes from the legend, one can easily identify how individual repartitioning objectives would be met.

From the plots, we can also identify the repartitioning choices for general-purpose OLTP application as *GR-MSM* and *GR-RM* followed by *CHR-MSM* and *CHR-RM*, while all of the *HR* based settings are highly tunable depending on the repartitioning objectives in response to different administrative situations. A key observation here is that, the choices of workload representation and mapping strategy are not bounded to any specific combination. To confirm this, two-way ANOVA test was conducted and later the interaction plots were analysed [27]. However, there was hardly any true evidence of interactions between the choices of representation and mapping strategy. Results from ANOVA table also support this finding. These series of observations strongly support the arguments presented in Sections 3 and 4, and justifies the goal of sensitivity analysis within a broad design space.

5. Experimental results

We evaluate the performance of the proposed incremental repartitioning methods using simulations. We design and develop

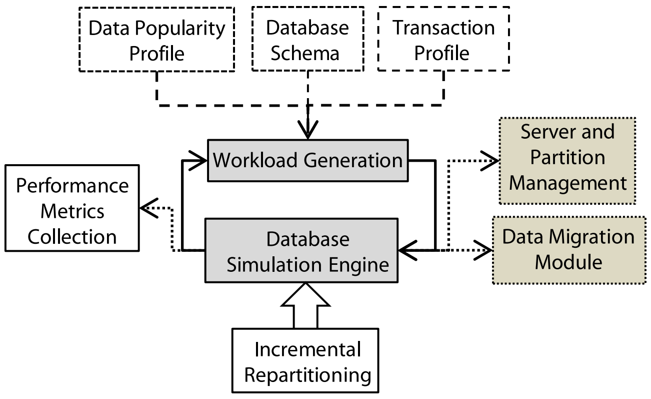

Data Popularity Profile
Database Schema
Transaction Profile
Workload Generation
Server and Partition Management
Performance Metrics Collection
Database Simulation Engine
Data Migration Module
Incremental Repartitioning

Fig. 7. An overview of the simulation framework.

a novel simulation platform using SSJ [28] capable of driving synthetic transactional workloads into distributed OLTP database cluster to evaluate different repartitioning algorithms. Using a parametric representation of the workload, underlying database, and system under test, we simulate a distributed OLTP database deployed in $|S|$ physical servers with a range or consistent-hash based initial data partitioning and supports workload-aware incremental repartitioning. Fig. 7 presents a high-level overview of the database simulation framework.

5.1. Workload and transaction profile

The simulation setup consist of distributed OLTP database and a workload generation process mimicking a scaled TPC-C [29] benchmark workload. A typical TPC-C database contains 9 tables, 5 transactions and simulates a order-processing transactional system within geo-graphically distributed districts and associated warehouses. The schema diagram for the considered TPC-C benchmark is shown in Fig. 8. Among the nine tables 'Stock' and 'Order-Line' tables grow faster in volume, thus all the logical database partitions are not homogeneous in size while 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. However, all the 90 partitions are distributed equally within the physical servers following a round-robin placement strategy thus ensures equal load at the very beginning of the database life cycle. The five transactions are weighted from heavy to light (in terms of processing) and occur in high to low frequencies. Two of the most high frequent transactions have strict response time requirements and covers almost 87.6% of the total generated transactions. The synthetic data generation process follows Zipf's law for generating 'Warehouse' and 'Item' tables' data and use the table relationship to generate others. We use the transaction generation model described at Section 4.7 with a parameter combination of $\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p\rangle = \langle 1, 0.25, 3600, 0.15\rangle$ for simulation purpose.

5.2. Simulation setup

We use both range and consistent-hash based initial data partitioning to evaluate four different scenarios to evaluate the proposed incremental repartitioning methods. Table 4 lists the related simulation parameters. Note that, tuple-level replication is not use in these settings as discussed in Sections 1 and 4. We simulate *range* and *consistent-hash* partitioned OLTP databases following two strategies – (1) no repartitioning or static (i.e., a single) repartitioning, and (2) incremental repartitioning – hourly or based on a defined threshold. We consider *No Repartitioning* *(NR)* and *Static Repartitioning (SR)* schemes 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 following the end of initial warm-up period and remain static for the remaining of its lifetime. As a proactive approach, *HR* repartition the database at the end of each observation window regardless of whether the value of I_d is below the threshold or not. This eventually reveals the highest ability of any data migration strategy to reduce the impact of distributed transactions in a successive manner. Finally, *TR* works upon an user defined value of I_d and only reacts when the per transaction per unit time I_d increases over the set threshold. Furthermore, we use *metis* [30] and *hmetis* [31] *k*-way min-cut clustering libraries with their default settings. These comprehensive analyses help to understand the applicability of incremental database repartitioning within a wide varieties of settings. We aim to evaluate the effectiveness of the proposed techniques with respect to—I_d, L_b, and D_m (as detailed in Section 4.8) for consecutive incremental repartitioning cycles based on (7), (10), and (11). Hence, we do not compare the results against the *performance measures* like transactional throughput and latency.

5.3. Result analysis

We simulate OLTP databases with two different initial data partitioning schemes—(1) *range*, and (2) *consistent-hash*. We adopt the following definition of *range* and *consistent-hash* schemes for

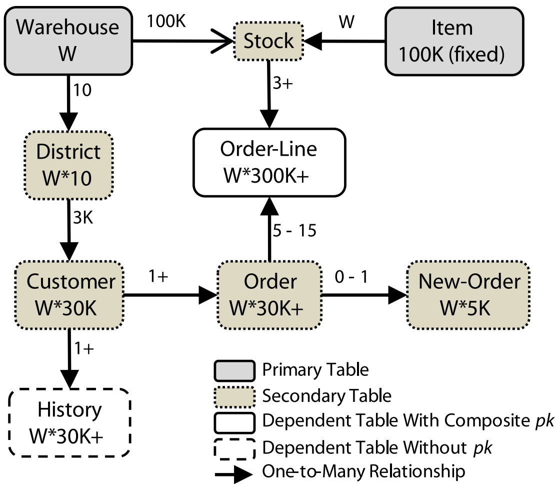


Fig. 8. Schema diagram of the TPC-C workload profile.

Table 4
Key simulation parameters.

Workload	TPC-C (Scale: 0.01)
Database tables	9
Number of physical servers, S	4
Partition assignment to server	Round-robin
Initial data partitioning	Range (36 Partitions) Consistent-hash (16 Partitions)
Workload modelling	GR, HR, CHR
Cluster-to-Partition mappings	RM, MCM, MSM
$\langle \mathcal{R}, \mathcal{U}, \mathcal{W}, p\rangle$	$\langle 1, 0.25, 3600, 0.15\rangle$
User defined I_d threshold	$2/S$
Smoothing factor for $\tilde{r}$, α	0.4
Compression level, C_l	6
Repartitioning schemes	NR, SR, HR, TR
Database warm-up period	03 h
Database operational period	24 h

initial data partitioning in the simulation. *Range* scheme partition each database table into a number of logical partitions with fixed volume size based on the total number 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 purpose. *Consistent-hash*, on the other hand, starts with a predefined number of logical partitions with very large data volume limit. For example, a consistent-hash ring made with 'SHA-1' has a bit space 2^{160} and having 16 fixed logical partitions each of them can hold up to $2^{160}/16$ data rows of a partitioned database.

In terms of scalability management, a new physical server can be easily added by redistributing only a fixed amount of data tuples within the cluster while *consistent-hash* scheme is in use. Data lookup process is also distributed and highly scalable as the consistent-hash function never changes. On the contrary, a centralised lookup server is required for *range* scheme and large volume of data migrations are required while adding a new server in the system. By adopting the proposed distributed data lookup method based on the concept of *roaming* (detailed in Section 4.6), we manage to include both of these data partitioning schemes into the simulation model. To clearly distinguish the difference between different incremental repartitioning schemes and data migration strategies we do not consider any workload variation during the lifetime of the simulated database. We also disable replication at the tuple level so that the actual performance of graph or hypergraph min-cut based clustering can be revealed. We compare four different repartitioning scenarios in the conducted experiments. Without adopting any repartitioning scheme at all the underlying database maintains a steady I_d over its lifetime. By applying any repartitioning scheme within this steady state causes data redistribution to minimise the value of I_d, and our goal is to find out how a static, proactive and reactive approach fit into this decision space. We examine the performance of 9 different incremental repartitioning schemes under 4 categories for both *range* and *consistent-hash* based partitioned databases. While setting the user-define threshold for I_d we prefer to set it as $|S|/2$ as it also means the target threshold should be such that, on average each transaction only spans half of the physical servers thus maintains a balance for physical resource consumptions.

5.3.1. *Range partitioned database*

Fig. 9(a) demonstrates the impact of distributed transactions for *range* partitioned database. As we observe, the value of I_d remains around 0.7 for the entire lifetime of the database if no repartitioning scheme is applied. This is the worst-case scenario according to the natural transactional properties of the underlying database. A single repartitioning step can hold down the rise of I_d for a certain amount of time, after that it gradually touches the margin again. Surprisingly, for *RM* data migration techniques and *GR-MSM* the increase of I_d is beyond 0.7 mark which also indicates that the *SR* is rather worse than having no repartitioning at all for some cases. Both *HR* and *TR* strategies for *RM* and *MSM* based 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 this particular workload pattern. *MCM* based strategies are the only incremental repartitioning technique that able to hold down I_d below the threshold margin. As *MCM* strategy tries to gather similar data tuples within a single server over the lifetime, *HR* tends to decrease I_d up to its lower bound. *TR* for *MCM* strategies, on the other hand, works as anticipated and perform incremental repartitioning whenever I_d suppresses the user-defined threshold level. For all the *MCM* strategies under this scheme have 9 incremental repartitioning cycles over the period of 24 h.

The server-level load-balance and inter-server data migration statistics are presented in Fig. 10(b) using (6) and (7). From the results it is clear that both *RM* and *MSM* based strategies maintain the load-balance for all workload representation types for both *HR* and *TR* schemes. In case of *MCM*, severe load-imbalance occurs for *HR* scheme because of the tendency to migrate more and more data tuples into the same physical machine over the successive repartitioning cycles. Initial *Range* partitioning also produces uneven sized logical partitions which also play an inherent role here as we make the decision of data migration at partition level. *TR*, on the other hand, perform better and the mean values for all workload representations are very close to *SR* technique.

In terms of inter-server data migrations, both *RM* and *MSM* strategies perform very similarly although the later exhibits slightly less movements than the former. For all the cases of *MCM*, data migrations kept very low although *TR* scheme has a higher range fluctuations comparing to *HR*. In overall, the proposed *MCM* strategy works very well for *Range* partitioned OLTP database that adopts *TR* in terms of reducing the overall impact of distributed transaction, maintain adequate server-level load-balance, and minimum inter-server data migrations.

5.3.2. *Consistent-hash partitioned database*

We compare the overall impact of distributed transactions in a consistent-hash partitioned OLTP database in Fig. 10(a). While *NR* scheme is applied, 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 Fig. 9(a) and is expected. Although *SR* reduces the margin of I_d in between 0.7 0.75 for both *RM* and *MSM* strategies while in between 0.65 0.7 for *MCM*, but its a matter of time when I_d gradually increase to reach the *NR* margin again. *RM* and *MSM* strategies perform similarly while *MSM* tends to indicate slightly better results. However, both of them fail to reduce I_d below the user-defined threshold limit for both *HR* and *TR* schemes. Similar to the *Range* partitioned database, *MCM* strategy performs well for all of the workload representations. Although, on average 15 repartitioning cycles are required for *TR* scheme comparing to the results of Fig. 9(a), but in overall *MCM* is the only strategy that is capable of holding any preset I_d threshold by the system administrator. *HR* scheme also perform similar to that of Fig. 9(a) and continue to reduce I_d in incremental repartitioning cycles for all workload representation types.

While analysing server-level load-balance as shown in Fig. 10(b), both *RM* and *MSM* strategies work very strictly in maintaining the ridge load-balance adopted by the initial consistent-hash data partitioning for all the cases. Although *MCM* strategies increases the flexibility of being *balanced*, however, the *TR* scheme continues to show better control over load-imbalance than the *proactive HR* approach. In terms of inter-server data migrations, *MSM* strategies perform slightly better than the *RM* ones, however both of them require a good amount of physical data migrations between the server to maintain a strict load-balance criteria. On the other hand, the flexibility adopted by *MCM* set it apart and by performing less data migrations thus adopting slight load-imbalance it can still reduce the impact of distributed transactions in a controlled manner. In comparing the *reactive TR* scheme with the *proactive HR* one, inter-server data migrations are higher in the former.

5.4. *Remarks on experimental results*

From the above discussed experimental analysis it is clear that *TR* based *MCM* strategy is better suited for all kind of databases—*range* or *consistent-hash* partitioned. In comparison with the *proactive* approach, *reactive* incremental cycles able to maintain a user-defined I_d threshold while keep constant check on the server-level load-balance with necessary amount of physical

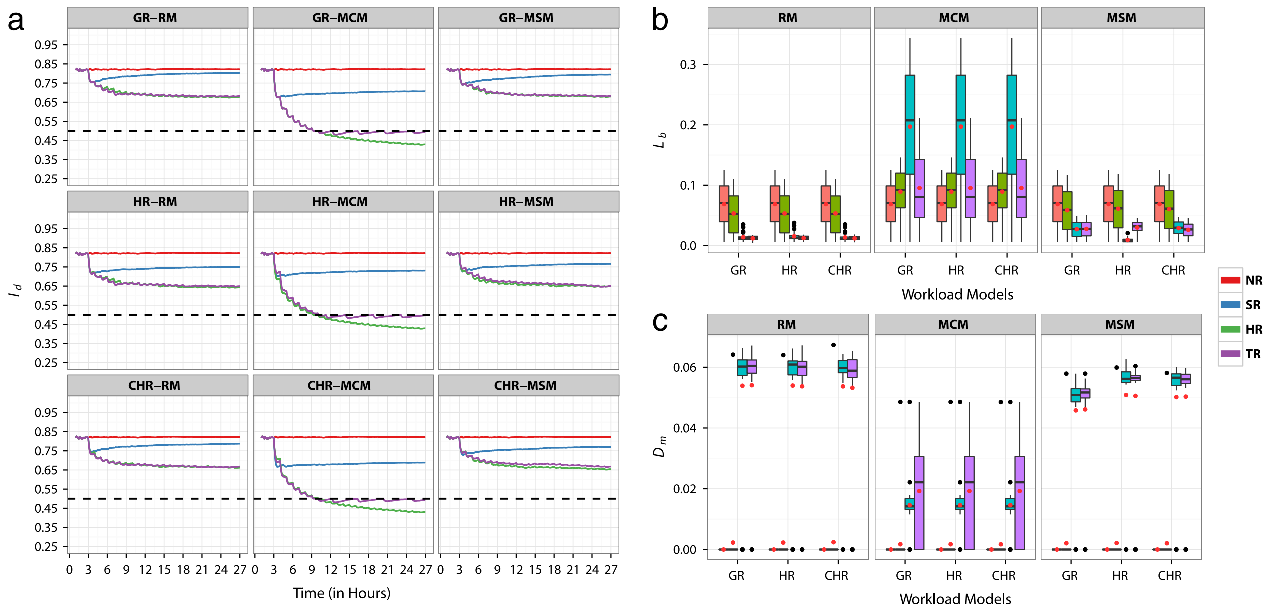


Fig. 9. Comparison of different incremental repartitioning schemes in a range partitioned shared-nothing OLTP database cluster for observing the variations of (a) impact of distributed transactions; (b) server-level load balance; and (c) inter-server data migrations.

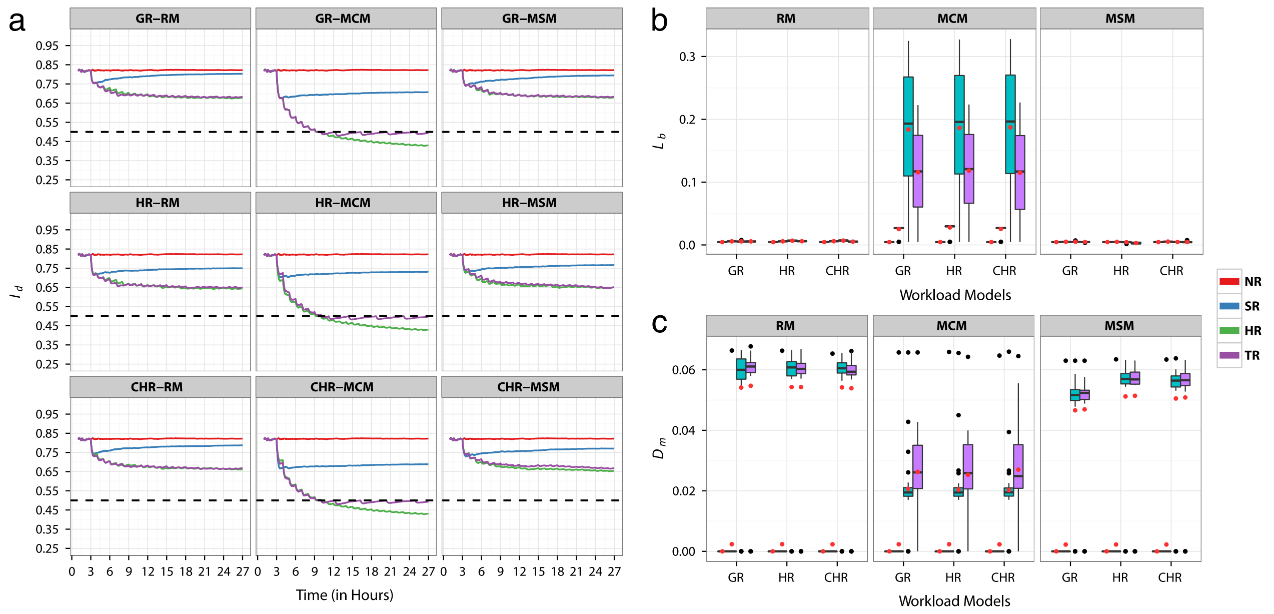


Fig. 10. Comparison of different incremental repartitioning schemes in a consistent-hash partitioned shared-nothing OLTP database cluster for observing the variations of (a) impact of distributed transactions; (b) server-level load-balance and (c) inter-server data migrations.

data migrations. In terms of workload representations, graph network representation is somehow ambiguous and not capable in producing the exact workload network, therefore, in sever cases the results are not consistent with the expected outcomes. For *HR*, although it represents the exact workload network but this provide more flexibility to the external min-cut libraries to produce such clusters which require higher amount of physical data migrations to maintain adequate load-balance. *CHR*, on the other hand, provide less flexibility for the external libraries as it combines inter-related hyper edges together and thus produce better results in reducing the overall impact of distributed transactions. In overall, *TR* based *CHR-MCM* incremental repartitioning shows better control and adaptability to both *range* and *consistent-hash* partitioned databases in a *reactive* manner.

6. Conclusions and future work

In this paper, we present a workload-aware incremental repartitioning framework for OLTP databases which minimises—(1) the impact of DTs using the *k*-way balanced min-cut clustering; (2) the overall load-imbalance through the randomness of the one-to-one cluster-to-partition mapping strategies; and (3) the physical data migrations by applying heuristics. Our innovative *transaction classification* technique ensures global minimisation in overall load-imbalance and data migrations comparing to the worst-case scenario of a Static Partitioning framework implementing random cluster-to-partition mapping for different workload representations. The *elaborate modelling* approach clearly identifies the inter-related goals within the repartitioning process, and provides effective heuristics to achieve them based on operational requirements. By adopting the concept of *roaming*, the proposed distributed data lookup technique transparently decentralise lookup operations from the distributed transaction coordinator guaranteeing high-scalability. Our philosophical arguments broaden the decision space with comprehensive *sensitivity analysis* by combining different workload representations and mapping strategies. The proposed set of *quality metrics* presents a sophisticated way to measure the quality of successive repartitioning, and the use of *composite metric* shows an effective way of operational intelligence for OLTP applications suffering from dynamic workload behaviours. For seamlessly evaluate the performance of any graph min-cut based incremental repartitioning we develop a generic transaction generation model for simulating OLTP partitioned databases which ensures avoiding any superfluous effects of external clustering libraries. Later, our detail sensitivity analysis and comparing the proposed methods in both range and consistent-hash based partitioned databases show the effectiveness of incremental repartitioning for achieving scalable transactional processing for modern OLTP applications. Our future works include—(1) analysis of workload networks to understand the underlying community structures for effective repartitioning, (2) utilise data stream mining for improved transaction classification and association rule based incremental repartitioning.

References

[1] IBM Data Hub, www.ibmbigdatahub.com/infographic/four-vs-big-data [Online: Last accessed 2015-03-02].
[2] L. Gu, D. Zeng, P. Li, S. Guo, Cost minimization for big data processing in geo-distributed data centers, IEEE Trans. Emerging Top. Comput. 2 (3) (2014) 314–323.
[3] R. Johnson, I. Pandis, A. Ailamaki, Eliminating unscalable communication in transaction processing, VLDB J. 23 (1) (2014) 1–23.
[4] J. Gray, A. Reuter, Transaction Processing: Concepts and Techniques, first ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.
[5] C. Curino, E. Jones, Y. Zhang, S. Madden, Schism: a workload-driven approach to database replication and partitioning, Proc. VLDB Endow. 3 (1–2) (2010) 48–57.
[6] A. Quamar, K.A. Kumar, A. Deshpande, SWORD: scalable workload-aware data placement for transactional workloads, in: Proceedings of the 16th International Conference on Extending Database Technology, EDBT'13, ACM, NY, USA, 2013, pp. 430–441.
[7] Vitess—scaling MySQL databases for large scale web services, https://github.com/youtube/vitess [Online: Last accessed 2015-03-02].
[8] MySQL toolkit for managing billions of rows and hundreds of database machines. https://github.com/tumblr/jetpants [Online: Last accessed 2015-03-02].
[9] A flexible sharding framework for creating eventually-consistent distributed datastores. https://github.com/twitter/gizzard/ [Online: Last accessed 2015-03-02].
[10] Apache Giraph. http://giraph.apache.org/ [Online: Last accessed 2015-03-02].
[11] P. Scheuermann, G. Weikum, P. Zabback, Data partitioning and load balancing in parallel disk systems, VLDB J. 7 (1) (1998) 48–66.
[12] M. Mehta, D.J. DeWitt, Data placement in shared-nothing parallel database systems, VLDB J. 6 (1) (1997) 53–72.
[13] K.A. Kumar, A. Quamar, A. Deshpande, S. Khuller, Sword: workload-aware data placement and replica selection for cloud data management systems, VLDB J. 23 (6) (2014) 845–870.
[14] A. Pavlo, C. Curino, 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, SIGMOD'12, ACM, NY, USA, 2012, pp. 61–72.
[15] T. Rafiq, Elasca: Workload-aware elastic scalability for partition based database systems (Master's thesis), University of Waterloo, Canada, 2013, http://uwspace.uwaterloo.ca/handle/10012/7525.
[16] P. Tözün, I. Pandis, R. Johnson, A. Ailamaki, Scalable and dynamically balanced shared-everything oltp with physiological partitioning, VLDB J. 22 (2) (2013) 151–175.
[17] I. Pandis, R. Johnson, N. Hardavellas, A. Ailamaki, Data-oriented transaction execution, Proc. VLDB Endow. 3 (1–2) (2010) 928–939.
[18] J.M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, P. Rodriguez, The little engine(s) that could: Scaling online social networks, IEEE/ACM Trans. Netw. (TON) 20 (4) (2012) 1162–1175.
[19] A. Turk, R.O. Selvitopi, H. Ferhatosmanoglu, C. Aykanat, Temporal workload-aware replicated partitioning for social networks, IEEE Trans. Knowl. Data Eng. 26 (11) (2014) 2832–2845.
[20] B.P. Swift, Data placement in a scalable transactional data store (Master's thesis), Vrije Universiteit, Amsterdam, Netherland, 2012, http://www.globule.org/publi/DPSTDS_master2012.pdf.
[21] Roaming in GSM Network. http://en.wikipedia.org/wiki/Roaming [Online: Last accessed 2015-03-02].
[22] M. Nicola, M. Jarke, Performance modeling of distributed and replicated databases, IEEE Trans. Knowl. Data Eng. 12 (4) (2000) 645–672.
[23] X/Open XA. http://en.wikipedia.org/wiki/X/Open_XA [Online: Last accessed 2015-03-02].
[24] Open Database Connectivity. http://en.wikipedia.org/wiki/Open_Database_Connectivity [Online: Last accessed 2015-03-02].
[25] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, 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, STOC'97, ACM, NY, USA, 1997, pp. 654–663.
[26] Mobile IP. http://en.wikipedia.org/wiki/Mobile_IP [Online: Last accessed 2015-03-02].
[27] J. Kamal, M. Murshed, R. Buyya, Workload-aware incremental repartitioning of shared-nothing distributed databases for scalable cloud applications, in: 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing, UCC, 2014, pp. 213–222.
[28] SSJ: Stochastic Simulation in Java. http://simul.iro.umontreal.ca/ssj/indexe.html [Online: Last accessed 2015-03-02].
[29] TPC-C—on-line transaction processing benchmark. http://www.tpc.org/tpcc/ [Online: Last accessed 2015-03-02].
[30] G. Karypis, V. Kumar, Multilevel k-way partitioning scheme for irregular graphs, J. Parallel Distrib. Comput. 48 (1) (1998) 96–129.
[31] G. Karypis, V. Kumar, Multilevel k-way hypergraph partitioning, in: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference, DAC'99, ACM, NY, USA, 1999, pp. 343–348.

Joarder Kamal received his Bachelor of Science in Computer Science and Engineering degree from Military Institute of Science and Technology, University of Dhaka, Bangladesh in 2008. He received his Masters by Research in Computing Science degree from Staffordshire University, United Kingdom in 2011. He also worked as a telecommunication engineer for network design and integration between 2008 to 2012. He is currently a final year Ph.D. student at the Faculty of Information Technology, Monash University, Australia. His research interests include large-scale distributed systems, big data analytic, machine learning, and mobile computing.

Manzur Murshed received the B.Sc. Engg. (Hons) degree in computer science and engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh, in 1994 and the Ph.D. degree in computer science from the Australian National University, Canberra, Australia, in 1999. He is currently a Robert HT Smith Professor and Personal Chair at the Faculty of Science and Technology, Federation University Australia. His major research interests are in the fields of video technology, information theory, wireless communications, distributed computing, and security and privacy. He has so far published 200 refereed research papers and received more than $1M nationally competitive research grants.

Rajkumar Buyya is a Fellow of IEEE, Professor of Computer Science and Software Engineering, Future Fellow of the Australian Research Council, and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercialising its innovations in Cloud Computing. He has authored over 450 publications and four text books. He is one of the highly cited authors in computer science and software engineering worldwide (h-index = 92, 41400+ citations). He has served as the foundation Editor-in-Chief (EiC) of IEEE Transactions on Cloud Computing and now serving as Co-EiC of Journal of Software: Practice and Experience.