Throughput Optimization in a Distributed Database System via Hypergraph Partitioning

Master's Thesis of

Patrick Firnkes

at the Department of Informatics
Institute of Theoretical Informatics, Algorithmics II

Reviewer:	Prof. Dr. Peter Sanders
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	M.Sc. Tobias Heuer
Second advisor:	M.Sc. Sebastian Schlag

1. November 2018 – 30. April 2019

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Walldorf, 30.04.2019

. . . . . .

(Patrick Firnkes)

Abstract

In this thesis, we introduce a workload-aware reassignment framework that optimizes the throughput of the distributed database *SAP Vora* for OLTP workloads. In distributed databases the primary way to scale is to horizontally partition the tables into *shards* and assign them across to hosts of a cluster. However, *distributed queries* are expensive in OLTP settings [8, 26]. Therefore, we optimize the shard assignment in a way that distributed queries are minimized while the load across the hosts of the cluster stays balanced.

Our approach monitors the executed queries, creates a hypergraph model of the workload and partitions it using the state-of-the-art hypergraph partitioner *KaHyPar* to find an optimized shard assignment. We integrate this reassignment framework into the distributed database SAP Vora. To the best of our knowledge, this is the first time that such a framework is implemented and evaluated in a commercial enterprise database system.

Furthermore, we present our novel approach *KaDaRea* that combines partitioning results from different time slices to get a better picture of the workload over time. Thus, it is able to optimize the assignment for workload peaks and changing workload patterns.

Finally, we evaluate our results using the popular *TPC-C* and *TPC-E* benchmarks which show that optimizing the shard assignment using our framework results in a great performance improvement in terms of throughput and response time, e.g. it increases the throughput for the TPC-E benchmark up to 5 times compared to the current state in Vora. Also, the evaluation shows that KaDaRea outperforms other state-of-the-art approaches as it increases the throughput by 42% during peaks and up to 92% if the workload patterns change compared to other approaches.

Zusammenfassung

In dieser Thesis wird ein arbeitslast-bewusstes Neuzuordnungs-Framework vorgestellt, das den Durchsatz der verteilten Datenbank *SAP Vora* für OLTP-Arbeitslasten optimiert. Hauptsächlich werden verteilte Datenbanken skaliert, indem die Tabellen horizontal in *Shards* partitioniert und über die Hosts eines Clusters verteilt werden. Jedoch sind *verteilte Anfragen* im OLTP Umfeld teuer [8, 26]. Deshalb optimieren wir die Shard-Zuordnung so, dass verteilte Anfragen minimiert werden und die Last über die Hosts des Cluster balanciert ist.

Unser Ansatz überwacht die ausgeführten Anfragen, erstellt ein Hypergraph-Modell der Arbeitslast und partitioniert das Modell mittels des State of the Art Hypergraph-Partitionierers *KaHyPar*, um eine optimierte Shard-Zuordnung zu finden. Wir integrieren dieses Neuzuordnungs-Framework in die verteilte Datenbank SAP Vora. Nach unserem besten Wissen ist dies das erste Mal, dass ein solches Framework in ein kommerzielles enterprise Datenbank-System integriert und evaluiert wird.

Darüber hinaus präsentieren unseren originellen Ansatz *KaDaRea*, der Partitionierungs-Ergebnisse von verschiedenen Zeitabschnitten kombiniert und so ein besseres Bild der Arbeitslast über die Zeit erhält. Dadurch ist es ihm möglich die Zuordnung für Lastspitzen und Änderungen in den Arbeitslast-Mustern zu optimieren.

Schließlich evaluieren wir unsere Ergebnisse mittels der weit verbreiteten *TPC-C* und *TPC-E* Benchmarks, die zeigen, dass durch das Optimieren der Shard-Zuordnung mit Hilfe unseres Frameworks eine große Performanz-Verbesserung in Bezug auf Durchsatz und Antwortzeit erreicht wird. So erhöht sich beispielsweise der Durchsatz für den TPC-E Benchmark um bis zu den Faktor 5 im Vergleich mit dem aktuellen Stand in Vora. Auch zeigen sie, dass KaDaRea bessere Ergebnisse als andere State of the Art Ansätze liefert, da KaDaRea, verglichen mit diesen, den Durchsatz um 42% für Lastspitzen und um bis zu 92% für Änderungen in den Arbeitslast-Mustern erhöht.

Contents

List of Figures

List of Tables

1. Introduction

In the age of *Big Data* and *Cloud Computing* databases are distributed over multiple physical machines called *hosts*. The primary way in which distributed databases are scaled is through horizontal partitioning of tables into *shards* and assigning these shards to hosts in the cluster [24]. This shard *assignment* (also called *allocation*) affects the performance of the system as depending on the assignment the hosts which participate in processing of a query change. A *distributed query* is a query that accesses shards which are placed on different hosts and thus multiple hosts participate in its processing.

In context of databases we can distinguish between *OLAP (On-line Analytical Processing)* workloads which consist of mostly long running queries for data analyzing and *OLTP (On-line transaction processing)* workloads which consist of mostly short running queries for executing daily business tasks [10]. In OLAP scenarios distributed queries are desired in order to distribute the load and parallelize the processing of a query. However, in OLTP scenarios distributed queries are expensive [8, 26] and should be kept to a minimum. In contrast to *OLAP* workloads, OLTP workloads are highly selective and fast running [10], thus they do not gain much from parallel processing on multiple hosts [8]. Therefore, we can improve the system performance by reducing overhead introduced by distributed queries. On the one hand, distributed queries lead to a communication overhead because the data must be exchanged between the involved hosts. On the other hand, they lead to a duplication of the load as a distributed query must be processed on each involved host, blocking processing slots for other queries. This decreases the performance in OLTP workloads, as there are many parallel queries creating a high load in the system, whereas in OLAP workloads there are only a single or few queries at a time [10]. Therefore, it is desired for OLTP workloads to minimize the number of distributed queries while balancing the load across hosts to substantially increase the transaction throughput [26].

To realize this, existing approaches place shards that are frequently accessed together on the same host while keeping the load of the hosts balanced. It is necessary to keep the load balanced as otherwise some hosts are overloaded while others are idling, leading to a performance degeneration [31]. This problem is called the *allocation problem*: For a given set of shards S and an expected query workload Q, the goal is to allocate the shards to hosts of a cluster such that a certain objective function for Q is maximized or minimized [26]. For instance, possible objective functions are the throughput of the system [8, 26] or the response time of queries [26, 31].

A common approach to solve the allocation problem is to use graph [8] or hypergraph [26] models of the workload and partition the model using graph or hypergraph partitioners to create an optimized shard assignment. In these workload models vertices represent shards and edges represent queries that co-access the spanned shards.

However, to the best of our knowledge, previous work did not integrate and evaluate their approaches in a commercial enterprise database system. Instead they implement

their approaches in less complex experimental research databases, like *Relational Cloud* or *H-Store* [8, 25]. Furthermore, previous work do not consider the workload over time when solving the allocation problem, instead they build the workload model as an aggregation over all queries. Therefore, they cannot detect changes in the intensity or mixture of the workload. Thus, they are not able to optimize for times where the workload is much higher, yet these are the times that require the best performing system. Also, without considering the workload over time it is not possible to detect patterns in the workload, e.g. created by multi-tenant databases. For instance, multiple teams from different regions are working on the same database. These teams access different parts of the database to execute their tasks which leads to two different patterns of workloads. Partitioning the workload model of these two workload patterns without considering the workload over time could result in an assignment that is bad for both patterns as it is imbalanced at any point in time.

1.1. Problem Statement

This thesis examines the question what the effects of introducing a *workload-aware reassignment approach* into the commercial enterprise database SAP Vora are and how the performance changes compared to the current state of assigning the shards.

Furthermore, this thesis investigates how a reassignment technique can consider the workload over time in order to react to changes in the workload patterns or to peaks in the workload.

1.2. Contributions

In this master's thesis we present a workload-aware database reassignment framework based on hypergraph partitioning to solve the allocation problem, which we integrated into the commercial enterprise database system SAP Vora.

Furthermore, we present our novel approach *KaDaRea* for solving the allocation problem that combines partitioning results from different time slices to get a better picture of the workload over time. This approach enables us to consider the time of execution of queries and therefore we can optimize the assignment for peaks in the workload and create more balanced and better performing assignments if the workload patterns changes.

Finally, we evaluate extensively the impact of using hypergraph partitioning to optimize the allocation of shards in a commercial enterprise database system by using the popular TPC-C and TPC-E benchmarks. The results show that the throughput increases up to 1.94 times for TPC-C benchmark and up to 5.11 times for the more complex TPC-E benchmark. Also, it shows that KaDaRea outperforms other state-of-the-art approaches as it increases the throughput by 42% during peaks and up to 92% if the workload patterns change compared to other approaches.

1.3. Outline

The outline of this thesis is a follows: Chapter 2 describes the tools and techniques used in this thesis, while Chapter 3 gives an overview on the related work. The approach and implementation of optimizing the assignment via hypergraph partitioning in SAP Vora are described in Chapter 4. Chapter 5 details the motivation and implementation of KaDaRea, our novel approach for solving the allocation problem. Our evaluation takes place in Chapter 6, in which we use the widely used TPC-C and TPC-E benchmarks to compare our results with the current state in Vora and other state-of-the-art approaches. Finally, Chapter 7 gives a conclusion on this thesis and Chapter 8 outlines the future work.

2. Foundations

This chapter gives an overview on the tools and techniques that create the basis for this thesis, including a definition of hypergraph partitioning and the allocation problem in context of distributed databases that is solved in this thesis. Furthermore, we present KaHyPar as a state-of-the-art hypergraph partitioner and the distributed database SAP Vora in which we integrate our solution.

2.1. Hypergraph Partitioning

2.1.1. Hypergraphs

Hypergraphs are a generalization of graphs where a *hyperedge* (also called *net*) can connect more than two *vertices* (also called *hypernodes*) [17]. Hypergraphs are classically applied in areas such as VLSI design to model circuits [17] or in scientific computing to compute sparse matrix-vector multiplications [9].

Definition 2.1 (Hypergraph). *An undirected weighted hypergraph $H = (V, E, c, \omega)$ is defined as a set of vertices V, a set of hyperedges E, where $\forall e \in E : e \subseteq V$, a hypernode weight function $c : V \to \mathbb{R}$, and a hyperedge weight function $\omega : E \to \mathbb{R}$.*

For a subset $V' \subseteq V$ and $E' \subseteq E$ we define:

$$c(V') = \sum_{v \in V'} c(v) \tag{2.1}$$

$$\omega(E') = \sum_{e \in E'} \omega(e)\,. \tag{2.2}$$

A hypergraph $H = (V, E)$ can be transformed into a graph by using the *clique* or *bipartite* transformation [15]. The clique transformation creates a clique graph $G_c = (V, E_c)$ where each hyperedge $e \in E$ is modeled as a clique between all vertices $u, v \in e$ with $u \neq v$. More formally, $E_c = \{\{u, v\} \mid \exists e \in E : u, v \in e \wedge u \neq v\}$. In contrast, the bipartite transformation creates a graph $G_b = (V \cup E, E_b)$ that models all vertices and hyperedges of H as nodes and connects each hyperedge e with an edge $\{e, v\}$ to all vertices $v \in e$. More formally, $E_b = \{\{e, v\} \mid \exists e \in E : v \in e\}$. The differences between these two transformations are shown in Figure 2.1.

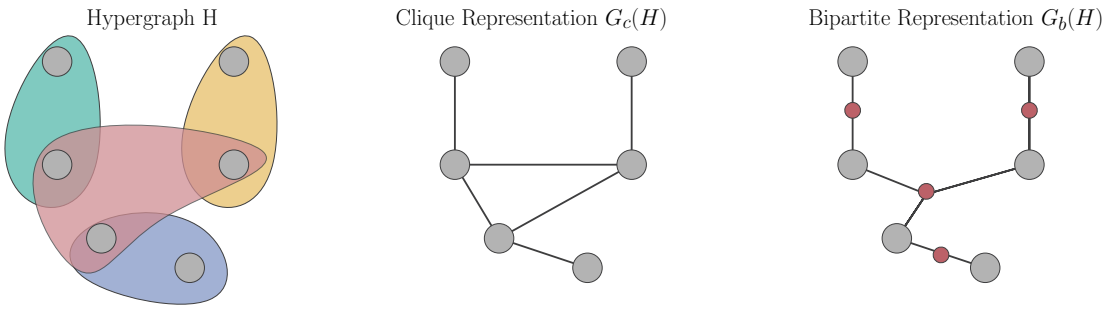


Figure 2.1: Clique and bipartite representation of a hypergraph

2.1.2. Problem Definition

Definition 2.2 (Hypergraph Partitioning Problem)**.** *The k-way hypergraph partitioning problem is to partition a hypergraph H into k disjoint non-empty blocks $\Pi = \{V_1, ..., V_k\}$ while minimizing an objective function on the nets and keeping a balance constraint so that all blocks are nearly equal sized regarding an imbalance parameter ε [13].*

A partitioning result Π is balanced if the following constraint is fulfilled:

$$\forall V_i \in \Pi \; : \; c(V_i) \leq (1 + \varepsilon) \cdot \left\lceil \frac{c(V)}{k} \right\rceil + \max_{v \in V} c(v) \, . \tag{2.3}$$

The *connectivity* of a hyperedge e is defined as $\lambda(e, \Pi) = |\{V_i \in \Pi \mid V_i \cap e \neq \emptyset\}|$, which is the number of blocks a hyperedge e is part of. A hyperedge e is cut if $\lambda(e, \Pi) > 1$. Further, we define $E(\Pi) = \{e \in E \mid \lambda(e, \Pi) > 1\}$ as the set of all cut hyperedges. There are two prominent objective functions in the hypergraph partitioning context: the *cut metric* and the *connectivity metric*. The cut metric is the generalization of the edge-cut objective in graph partitioning:

$$cut(\Pi) = \sum_{e \in E(\Pi)} \omega(e) \, . \tag{2.4}$$

The other prominent metric is the connectivity metric (also called $(\lambda - 1)$ metric) which considers how many blocks a hyperedge is spanning:

$$(\lambda - 1)(\Pi) = \sum_{e \in E} (\lambda(e, \Pi) - 1)\omega(e) \, . \tag{2.5}$$

Figure 2.2 shows a hypergraph that is partitioned into $k = 3$ blocks where all blocks have an equal size of $c(V_i) = 2$. The cut metric for this example is $cut(\Pi) = 1$, whereas the connectivity metric is $(\lambda - 1)(\Pi) = 2$.

In contrast to *vertex partitioning*, which was described in the previous part of this section, *hyperedge partitioning* partitions hypergraphs by cutting vertices instead of edges [32]. Yang et al. proposed a hypergraph partitioning algorithm for hyperedge partitioning based on hyperedge moves where a hypergraph $H = (V, E, c, \omega)$ is partitioned into k disjoint sets of hyperedges $\Pi = \{E_1, ..., E_k\}$ while keeping the weight of the block balanced.

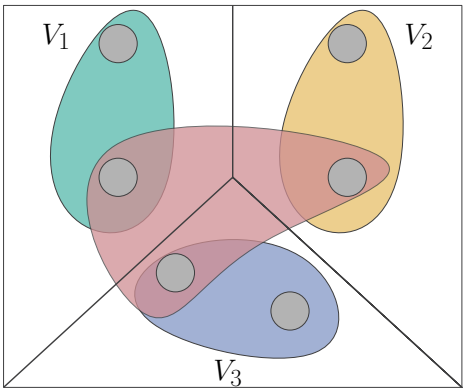


Figure 2.2: Hypergraph partitioned into $k = 3$ blocks

Analogously, we define $\lambda(v, \Pi) = |\{E_i \in \Pi \mid v \in E_i\}|$ as the number of blocks a vertex v is part of and $V(\Pi) = \{v \in V \mid \lambda(v, \Pi) > 1\}$ as the set of cut vertices.

Using these definitions the cut and connectivity metric can be defined as follows [32]:

$$cut(\Pi) = \sum_{v \in V(\Pi)} c(v) \tag{2.6}$$

$$(\lambda - 1)(\Pi) = \sum_{v \in V} (\lambda(v, \Pi) - 1)c(v)\,. \tag{2.7}$$

2.1.3. Multilevel Paradigm

Hypergraph partitioning is known to be NP-complete [20], therefore several heuristics have been created to solve the partitioning problem. The most common heuristic to solve the problem is the *multilevel paradigm* [27].

Such algorithms consist of three phases: The first phase is the *coarsening* phase in which the input hypergraph is recursively coarsened to create a hierarchy of smaller hypergraphs by calculating clusterings or vertex matchings, which are then contracted. Each coarsened hypergraph represents one *level*. As soon as a predefined number of vertices is reached, the *initial partitioning* phase takes place in which algorithms are applied to the smallest hypergraph to partition it into k blocks. Finally, in the *refinement* phase the coarsening is undone by uncontracting the vertices in reverse order of contraction and simultaneously using a *local search* heuristic to improve the quality of the solution [27]. Figure 2.3 depicts the process of a multilevel hypergraph partitioning.

KaHyPar[1] is a state-of-the-art hypergraph partitioner developed by researchers at Karlsruhe Institute of Technology. It takes the multilevel paradigm to its extreme by contracting only a single vertex in each level of the multilevel hierarchy. It supports *direct k-way* as well as *recursive bisection* partitioning. Direct k-way partitioning partitions the hypergraph directly into k balanced blocks, in contrast to recursive bisection where the hypergraph is recursively bipartitioned until k blocks are reached [1].

[1] `https://github.com/SebastianSchlag/kahypar`

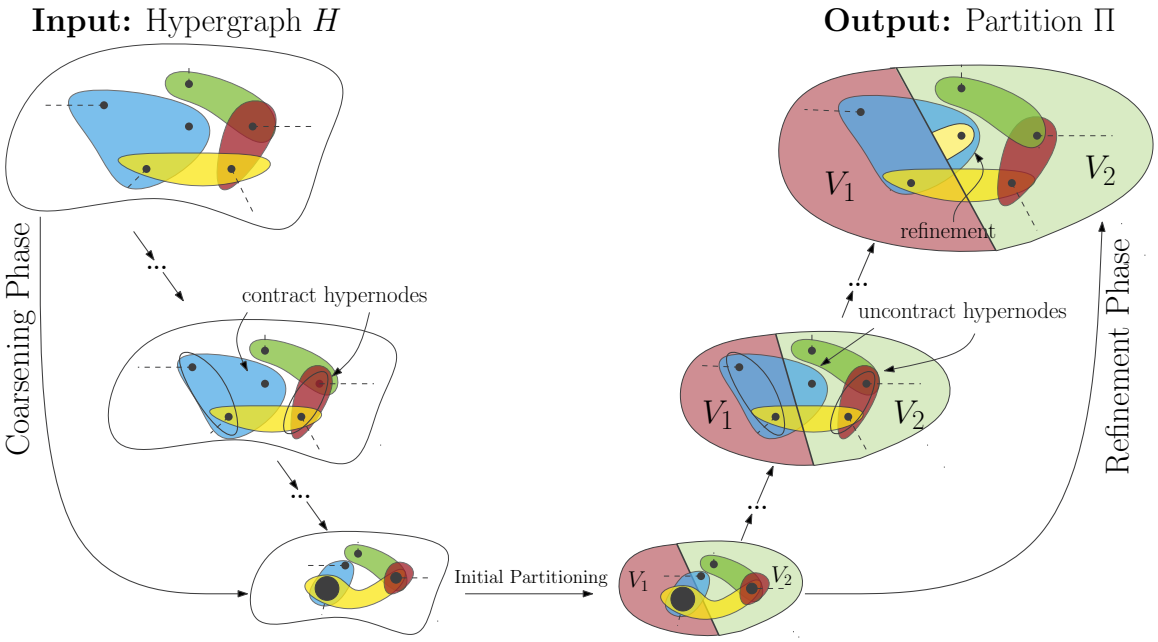


Figure 2.3: Multilevel hypergraph partitioning [12]

Furthermore, KaHyPar supports the cut as well as the connectivity metric as objective function for partitioning the hypergraph [1].

By applying it's n-level approach combined with strong local search heuristics, KaHyPar outperforms other state-of-the-art hypergraph partitioners in solution quality [14].

2.1.4. Maximum Weighted Bipartite Matching Problem

This section defines the *maximum weighted bipartite matching problem* as we make use of it later in this thesis.

Definition 2.3 (Bipartite Graph). *A bipartite graph $G = (V = L \cup R, E)$ with $L \cap R = \emptyset$ is a graph where $\forall e = \{u, v\} \in E : u \in L \wedge v \in R$.*

Based on the definition of bipartite graphs we can now define the *maximum bipartite matching problem* as:

Definition 2.4 (Maximum Bipartite Matching Problem). *Given a bipartite graph $G = (V = L \cup R, E)$, the maximum bipartite matching problem is to select a subset $M \subseteq E$ such that for each node $v \in V$ there is at most one edge $e \in M$ with $v \in e$ while maximizing $|M|$. If for each node $v \in V$ there exists exactly one edge $e \in M$ with $v \in e$, then M is called a perfect matching.*

This problem can be solved by converting the graph into a flow network and calculating the maximum flow [7]. An extension of the maximum bipartite matching problem is the *maximum weighted bipartite matching problem*:

Definition 2.5 (Maximum Weighted Bipartite Matching Problem). *Given a weighted bipartite graph $G = (V = L \cup R, E, \omega)$ with an edge weight function $\omega : E \to \mathbb{R}$, the maximum*

weighted bipartite matching problem is to select a bipartite matching $M \subseteq E$ such that the sum of edge weights in M is maximized.

This problem also appears in context of hypergraph partitioning with fixed vertices, which means that vertices are preassigned to specific blocks where they should be assigned to after the partitioning [2].

One method that solves the maximum weighted bipartite matching problem is the *Hungarian algorithm*, which makes use of the duality between the maximum weighted bipartite matching problem and finding the minimum weighted vertex cover in bipartite graphs [19].

Definition 2.6 (Minimum Weighted Vertex Cover Problem)**.** *Given a weighted bipartite graph $G = (V = L \cup R, E, \omega)$ with an edge weight function $\omega : E \to \mathbb{R}$, the minimum weighted vertex cover problem is to choose labels $U = (u_1, ..., u_{|L|})$ and $V = (v_1, ..., v_{|R|})$ such that $\forall i, j : u_i + v_j \geq \omega_{i,j}$ while minimizing the sum of all labels.*

The Hungarian algorithm creates such a vertex cover U, V for a bipartite graph G. It constructs a subgraph which contains an edge between each node $u \in L$ and $v \in R$ where for its corresponding labels the condition $u_i + v_j = \omega_{i,j}$ holds. If we find a perfect matching in this subgraph, we can return this matching as a solution for the problem. Otherwise, we adjust the cover until we find a perfect matching. For more implementation details we refer the reader to [19]. This algorithm solves the maximum weighted bipartite matching problem in $O(n^3)$ time [19].

Because KaHyPar supports hypergraph partitioning with fixed vertices, it contains an implementation of the Hungarian algorithm to solve the maximum weighted bipartite matching problem. We are reusing its implementation in this thesis.

2.2. Distributed Databases

A *distributed database* is a collection of multiple, logically interrelated databases (also called *hosts*) distributed over a computer network [24]. The data of a distributed database is partitioned using a partition function and spread across the hosts of the database. Distributed databases promise higher scalability, reliability, and availability than non-distributed databases, but require more complex mechanisms to guarantee data integrity [24] and there is a communication overhead in query processing because hosts have to communicate with each other to process queries that touch data from multiple hosts.

Firstly, we describe how distributed databases are partitioned into shards to make them scalable. Then we detail the distributed database SAP Vora as our approach is implemented in this system and lastly we describe the differences between OLTP and OLAP workloads.

2.2.1. Database Partitioning

The primary way in which distributed databases are scaled is by *horizontal partitioning* of the data [8]. Horizontal partitioning splits the rows of a table into disjoint subsets, called *shards* in this thesis. These shards are then distributed over the hosts of the database.

Two prominent partition functions for database partitioning are *hash partitioning* and the *range partitioning*.

Let $T = (t_1, ..t_n)$ be a table, where t_i represents tuple i of T and $p(t_i)$ be a function that extracts partition relevant information from row t_i. An example for such a function is the extraction of the value of the primary key from tuple t_i. We can now create a hash partition function by using a universal hash function $h(p(t_i))$ that assigns all tuples t_i to a shard with index $\{1, .., k\}$, where k is usually the number of hosts.

Instead of using a hash partition function, one could also use a range partition function. Based on a sequence $S = (s_1, ..., s_{k-1} \mid s_j < s_{j+1})$ the shards are created by assigning each row t_i to shard l where l is either the highest index for which $p(t_i) \leq s_l$ is fulfilled or k if $p(t_i) > s_{k-1}$.

The benefit of using a hash function is that it leads to shards of equal size and avoids clustering of the data [23]. On the other hand, range partitioning increases the performance for range scans on the partitioned column, hence it is often used to partition tables on a timestamp column, leading to small scans if data from a month or year is queried [23].

2.2.2. SAP Vora

SAP Vora is a distributed database system for big data processing and is built to scale with load by scaling up the number of computing nodes in the cluster [28].

To make the database scalable and to distribute the load, the tables are horizontally split into shards using hash or range partition functions and these shards are assigned to the computing nodes in round-robin fashion [28]. Vora is designed for both *OLTP* and *OLAP* workloads [11] (see Section 2.2.3 for the differences between OLTP and OLAP).

Figure 2.4 shows the architecture of Vora. The *transaction coordinator* is the entry point of the system, which can be used to execute queries by connecting to via *Vora Tools* or interfaces like *JDBC*. The transaction coordinator also controls the execution of queries by generating an *execution plan* which is send to the engines. For the plan generation it fetches metadata about the database structure and the data layout from the *catalog server* and host assignment from the *landscape manager*. After sending the execution plan to the engines, the engines generate execution code for the plan, execute the code to get the result whereby the engines communicate with each other if it is required and propagate the result back to the user. Vora supports different kind of engines like *in-memory* or *disk engines*. Other important components are the landscape manager which is responsible for data placement and the *distributed log* which is responsible for persistence of metadata and information needed to recover system after failover.

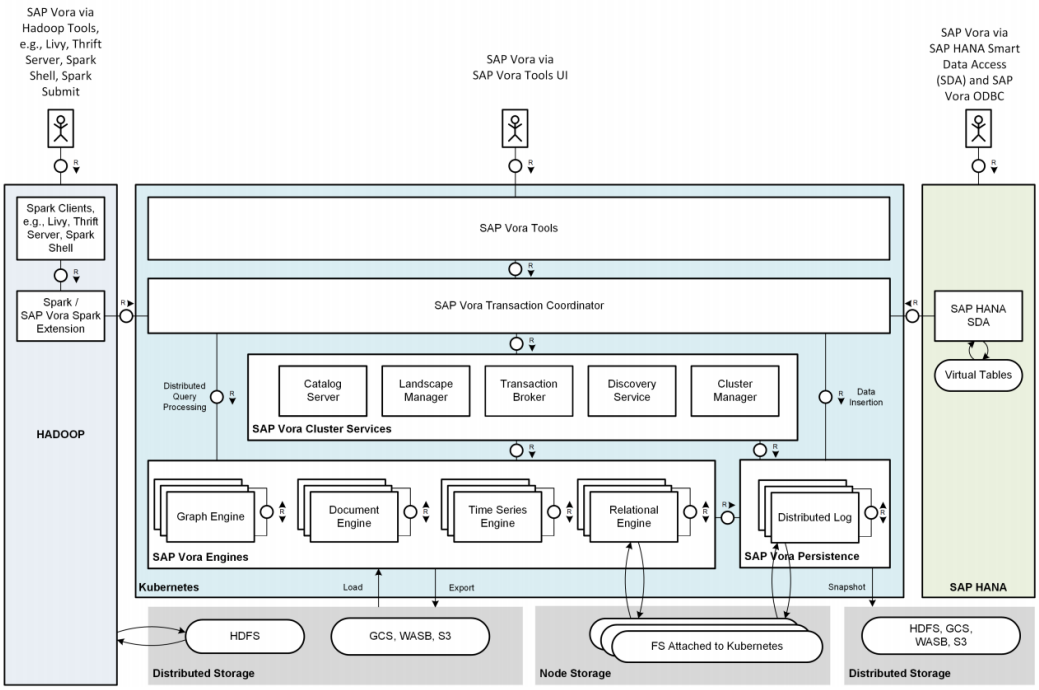


Figure 2.4: SAP Vora architecture [28]

2.2.3. OLTP vs. OLAP Workloads

On-line transaction processing (OLTP) workloads are characterized by many short running queries that often affect only single or a small number of tuples in the database [10]. The workload consists of `SELECT`, `INSERT`, `DELETE`, and `UPDATE` statements, which are used to control and run daily business tasks [10]. Because the queries are typically rather short running, systems designed for OLTP workloads try to optimize the throughput of the system [10].

On the other hand, *on-line analytical processing* (OLAP) workloads consist of long running queries with a low volume of transactions that often affect a large number of tuples in the database [10]. The workload consists of read-only queries which are operating on consolidated data from one or several OLTP databases [5]. These workloads are used to analyze data and help with decision making and planning. For OLAP systems the response time of queries is a good measuring metric [10].

2.3. Allocation Problem

In general the resource allocation problem is to find an optimal allocation of a fixed amount of activities to resources so that the cost incurred are minimized [18].

This thesis solves the allocation problem in context of distributed databases and OLTP workloads. We can define the allocation problem in this context as:

Definition 2.7 (Allocation Problem)**.** *Given a distributed database consisting of a set of hosts $H = \{1, .., k\}$, a set of shards $S = \{s_1, .., s_n\}$, an expected query workload $Q = (q_1, ..., q_m)$, the allocation problem is to assign shards to the hosts such that a certain objective function for the given workload is maximized or minimized.*

For instance, possible objective functions are the throughput of the system [8, 26] or the response time of the queries [26, 31].

In distributed database tables are partitioned into shards and spread across the hosts of the database. Queries that touch shards from multiple hosts are expensive in OLTP settings because they lead to a communication overhead and a duplication of the load [8]. Furthermore, it should be aimed to balance the load across the hosts to avoid overloaded or idling hosts.

3. Related Work

The following sections give an overview on related workload-aware database partitioning approaches.

3.1. Schism

Schism [8] is an approach using graph partitioning to partition tables in distributed databases for OLTP workloads implemented in the experimental database *relational cloud*. Besides partitioning, it also implements replication of tuples to avoid distributed transactions. Furthermore, it provides an explanation phase using machine learning techniques that tries to find a simple model for the mappings produced in the partitioning phase. In this explanation phase it creates a decision tree and decides if the found partitioning is either equal to hashing, range-predicates or look-up tables.

Schism logs the executed queries, generates `SELECT` queries to find the involved tuples which causes a second execution of the workload, and using these found tuples to build the workload graph [8].

It models a tuple of a table in the database as a vertex in the graph and vertices that are connected by an edge represent tuples that are co-accessed. The weight of an edge describes the frequency of co-accessing the connected vertices and the vertex weight describes how often a tuple is accessed. Figure 3.1 depicts this graph model for an example workload. To find the best partitioning Schism minimizes the cut metric.

To reduce the distributed transactions even further, Schism introduces tuple-level replication by expanding a node into a star-shaped group of $n + 1$ nodes where n is the degree of the node. The edges between these nodes represent the replication cost of the tuple which is the number of updates affecting it, because in contrast to reads, updates are then distributed over multiple blocks. Finally, the tuple is replicated to each block that it is assigned to after the graph partitioning.

However, in contrast to hypergraphs, graphs cannot represent the executed transactions correctly, e.g. one cannot differentiate if two edges belong to the same transaction or if they represent two separate ones, whereas in context of hypergraphs a hyperedge clearly represents a single transaction. Also, Wagner et al. proved that there is no general model that can be used to transform a hypergraph $H = (V, H)$ into a graph $G = (V, E)$ that returns the same min-cut model [16]. The authors of Schism decided against the usage of a hypergraph to model the workload because the hypergraph partitioners at that time did not perform well [8].

In addition, the scalability of Schism is questionable because the vertices in the graph represent tuples and not an aggregation of them [26].

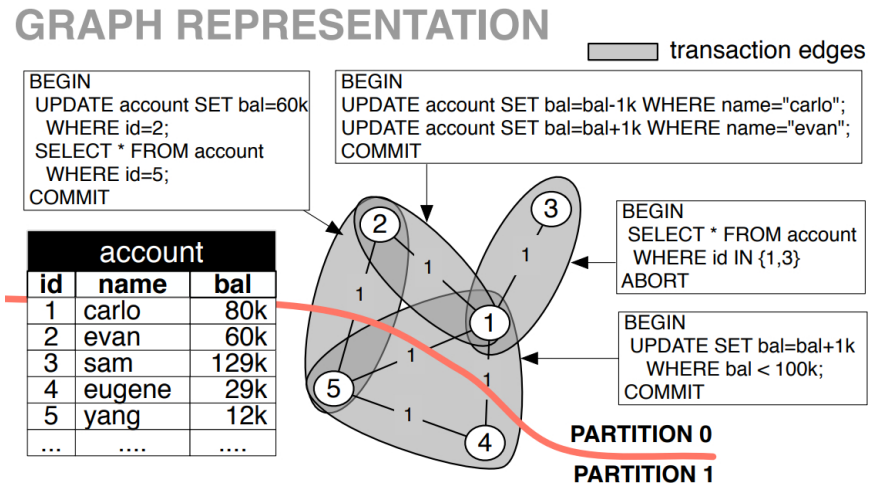


Figure 3.1: Schism's graph model of workload [8]

3.2. SWORD

SWORD [26] is based on hypergraph partitioning and focuses on scalability, tolerance to failures and workload changes. One technique used to improve scalability is the generation of a compressed hypergraph where multiple tuples in a table are collapsed into a single *virtual node* using a hash function. This technique is widely used in distributed databases to partition tables. SWORD is implemented in a transaction manager that orchestrates a number of PostgreSQL instances and routes incoming queries to their destination databases [26]. This transaction manager was specially created for the implementation of SWORD [26].

As Schism, SWORD is designed for OLTP workloads and it minimizes the cut metric to find the best partitioning. The hypergraph models the workload as $H = (V, E, c, \omega)$ where V represents the virtual nodes and a hyperedge $e \in E$ represents a transaction with its spanned nodes. In the workload model the weight of a node represents how often this node was accessed and the weight of a hyperedge represents the frequency of the transactions. Figure 3.2 shows the difference between a workload model based on tuples and based on compressed nodes.

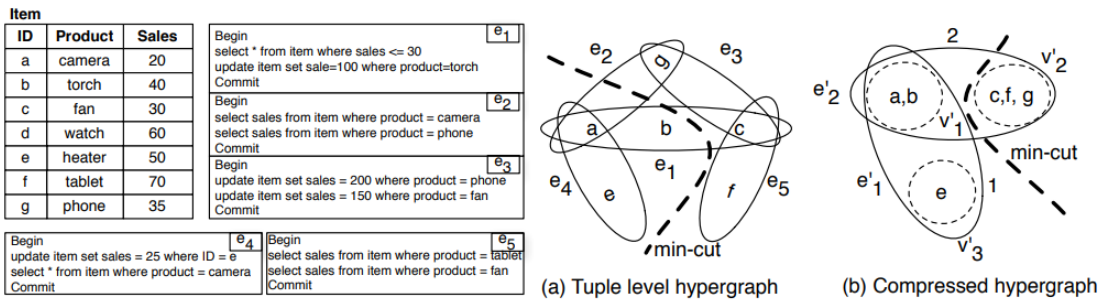


Figure 3.2: SWORD's hypergraph model of workload: (a) Tuple level hypergraph, (b) Compressed hypergraph [26]

SWORD proposes an incremental repartitioning technique to avoid resorting to complete data migration when the workload has changed. For that it monitors the number of distributed transactions and triggers a data migration step whenever a threshold is crossed.

In contrast to Schism, SWORD aggressively replicates the data and does not trade-off performance for fault tolerance, thus each tuple is at least replicated once. Depending on the write-to-read-ratio of a tuple it can be replicated multiple times - the smaller the ratio is, the more replicas are created to avoid distributed reads. One drawback of this aggressive replication is overhead introduced for distributed updates. To control this overhead *quorums* like *read-one-write-all* (updates need to access all replicas) or *majority* (updates need to access more than 50% of the replicas) are used, which control the number of blocks that must be accessed in a transaction. SWORD defines these quorums on virtual node level depending on their access pattern. However, another drawback of the replication is that data size is at least two times as big, which makes it questionable if SWORD can be applied for large data sizes.

SWORD follows Schism's approach of logging the executed queries, generating `SELECT` queries to find the involved tuples, and using them to build the hypergraph model [26].

3.3. Clay

In contrast to most other approaches, *Clay* [29] is an on-line partitioning approach for OLTP workloads that is based on a greedy algorithm instead of graph partitioning. It is implemented in the experimental research database *H-Store* and hooks into its database query processing components to build the workload graph, hence it does not require to query the tuples a second time for model building as Schism and SWORD do [29].

Clay builds a so called *heat graph* with the accessed tuples V where the weight of the vertices and edges E represent the frequency of access, which is basically the same model as Schism's workload graph. Clay uses this model to search for *hot* (frequently accessed) tuples and tuples that are co-accessed with them. These build a so called *clump* of tuples that should be placed together. Figure 3.3 shows how such a heat graph and an example clump look like.

Clay's migration algorithm starts by finding a set of overloaded blocks that have a load higher than a threshold θ which is the average load across all blocks multiplied with $1 + \varepsilon$. The load L_Π of block $V_i \in \Pi$ is defined as:

$$L_\Pi(V_i) = \sum_{v \in V_i} c(v) + \sum_{\substack{u \in V_i \\ v \notin V_i \\ \{u,v\} \in E}} \omega(\{u,v\}) \cdot k\,, \tag{3.1}$$

where k is a constant factor representing the cost of a distributed transaction (set to $k = 50$).

It then initializes a clump M by finding the hottest tuple h in an overloaded block and finding a target block d that minimizes the overall load of the system. M is then expanded with the most frequently co-accessed tuples of h. After each expand, Clay verifies that the move of M is still *feasible*, which means that d does not become overloaded or the *receiver delta* is negative. The receiver delta for a clump M to receiver d describes how the load of

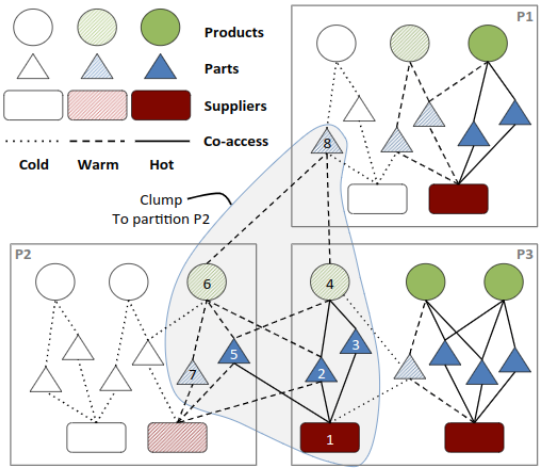


Figure 3.3: Clay's heat graph [29]

block d changes if M would be moved to it. Because the distributed transactions are part of the load definition, a negative receiver delta implies that the cut is reduced.

If the move is not feasible, it tries to find another destination block for M. Clay creates and moves clumps until no block exceeds the threshold θ or it cannot find a feasible move any more.

Clay works best with workloads that are highly skewed [29]. However, the authors of Clay claim that they substantially outperform other state-of-the-art workload-aware database repartitioning techniques like Schism [29]. Additionally, they claim to require less data migration than other approaches.

One drawback is that Clay stops as soon as all blocks have an equal load and does not continue until the overall load is minimized, which is shown in Figure 3.4. Tuples v_1 and v_3 are placed on one block and v_2 and v_4 are placed on another block. As v_1 is queried together with v_2 and v_3 is queried together with v_4, the number of distributed queries is maximal for the current assignment. However, the load for both blocks is equal, and thus the algorithm does not optimize it. This makes it questionable if Clay performs better than other partitioning approaches based on graph or hypergraph partitioning techniques.

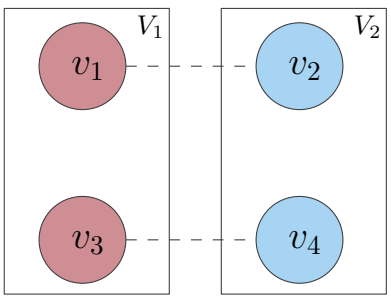


Figure 3.4: Heat graph that is balanced under Clay's load definition and thus no optimization takes place.

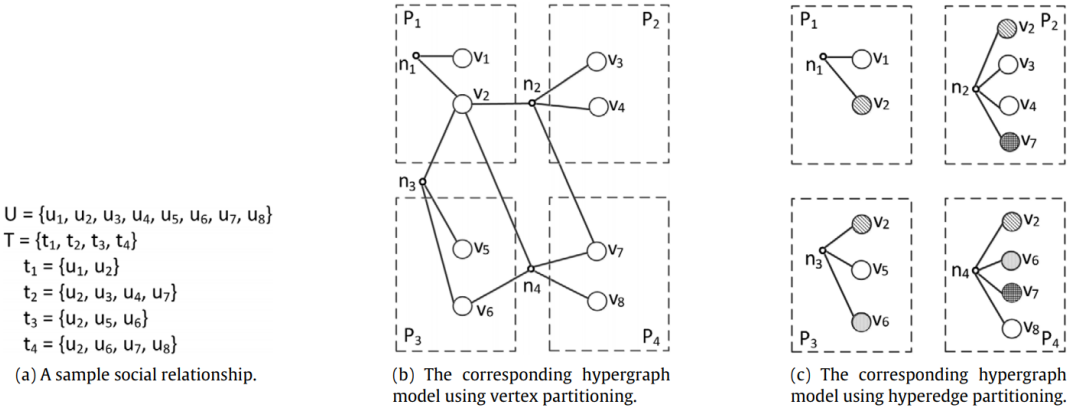


Figure 3.5: Difference between vertex and hyperedge partitioning [32]

3.4. HEPart

In contrast to other workload-aware database repartitioning approaches, *HEPart* [32] uses hyperedge partitioning instead of vertex partitioning.

Hyperedge partitioning has the advantage that vertices are replicated in the partitioning phase, and thus it does not require an additional modeling step over auxiliary vertices as other approaches. Figure 3.5 shows the difference between vertex and hyperedge partitioning: In hyperedge partitioning a hyperedge always belongs to only one block and a vertex belongs to one or more blocks if it is cut, whereas in vertex partitioning a vertex always belongs to one block and a hyperedge to one or more if it is cut. By using hyperedge partitioning HEPart reduces the distributed transactions by the greatest extent, but it could lead to a large number of replicas. However, we cannot apply a hyperedge partitioning as HEPart in Vora as it does not support replication yet.

HEPart is designed for general big data applications and could be applied to a model of database workload where hyperedges represent queries, vertices represent tuples in the database and the weight of them is related to their load. The gain $g_{pq}(e)$ of a hyperedge e refers to the reduction in the cut if e would be moved from block p to *block* q. To partition the hypergraph, HEPart computes the possible gains and pin distributions for each hyperedge. It then moves the hyperedge with the highest gain that does not violate the balance constraint if it is moved. After the move the gains and pin distributions of all hyperedges that have at least one common vertex are updated. It continues to move hyperedges until no move that decreases the cut and does not violate the balance constraint is possible. Finally, tuples are placed on a block if they are connected to at least on hyperedge in that block.

However, Yang et al. did not implement a prototype in a distributed database system and thus performance comparisons to other approaches are missing.

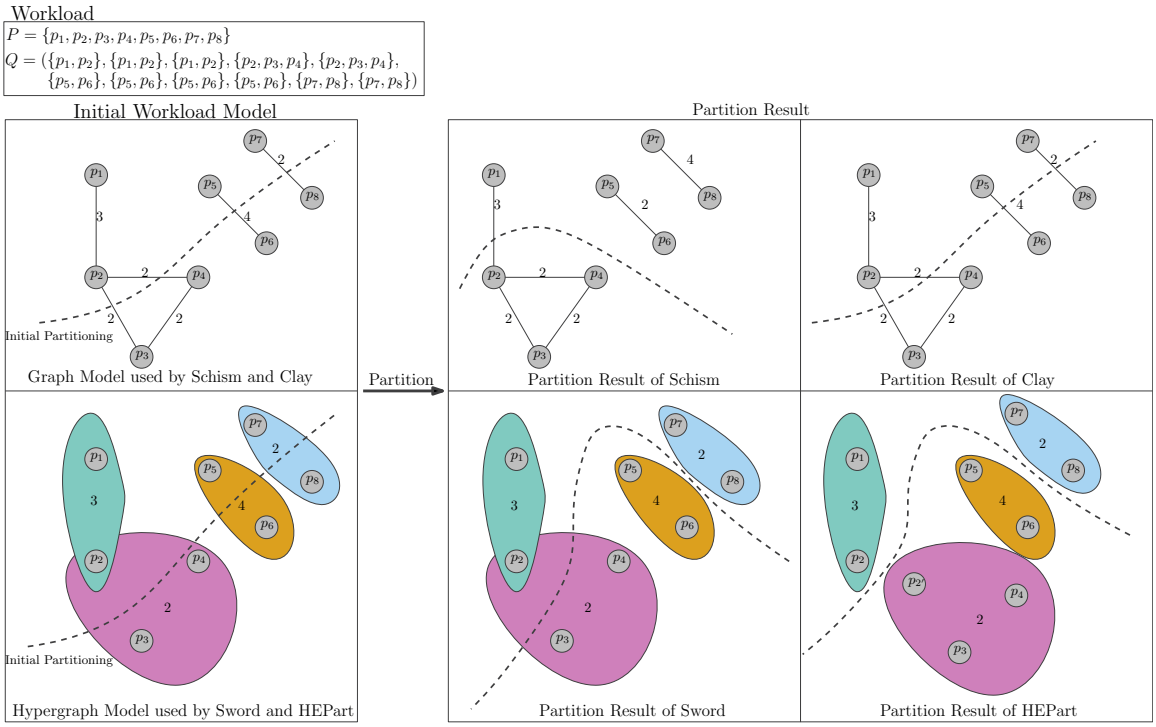


Figure 3.6: Comparison of model and partitioning result of Schism, SWORD, Clay and HEPArt

3.5. Comparison

This section compares the model and partitioning result of Schism, SWORD, Clay, and HEPart, whereby replication of shards is omitted. Figure 3.6 shows that Schism and Clay are using graph models whereas SWORD and HEPart are using hypergraph models to model the workload.

Using graph models instead of hypergraph models can lead to a worse partitioning results as the example shows. To reduce the number of distributed queries p_2 should be placed on the same block as p_1. However, Schism places these nodes on two separate blocks resulting in 50% more distributed queries than SWORD's result. That is the case, because in the graph model the smallest cut is 3 by cutting the edge between p_1 and p_2, whereas in the hypergraph model the smallest cut is 2 by cutting the hyperedge containing p_2, p_3, and p_4.

Furthermore, the figure shows that Clay does not change the partitioning, even if the number of distributed queries could be reduced greatly. That is the case because both blocks have nearly equal load according to their definition of load (see Equation (3.1)). Clay is designed to work best for skewed workloads and does not improve the partitioning in this case [29].

Lastly, HEPart creates a solution with no distributed queries by using hyperedge partitioning. However, this can only be reached by replicating p_2, which is currently not supported by Vora.

3.6. Other Approaches based on Hypergraph Partitioning

Beside SWORD, there are several other approaches based on hypergraph partitioning that differ in details and are only slightly related to our work. These approaches are briefly described in this section.

Catalyurek et al. presented a hypergraph model for periodic load balancing that does not only consider the communication costs of distributed transactions, but also the data migration costs [4]. Thus, the partitioning result optimizes the minimal total execution time needed for executing the transactions and migrating the data. To consider the migration cost in the partitioning they insert a fixed vertex called *partitioning vertex* for each block and connect the partition vertex with all data nodes that are assigned to it. The weight of these hyperedges represents the cost of moving the data which is equal to the size of it.

Yu et al. examined the data placement in geo-distributed applications and created an approach called *ADP* that considers the location of data and services when optimizing the data placement [34]. To realize this, ADP models two types of vertices: *storage nodes* that represent data centers and *data items* that represent items that should be assigned to the storage nodes. Storage nodes are connected with each data item and the weight of this hyperedge is the frequency of access from the data center. Also, data items that are accessed together are connected by a hyperedge with the frequency of access as weight. By placing each storage node in a separate block using fixed vertices, the partitioning result can be used to place the data items to their nearest data center, optimizing localized data serving and the co-location of associated items.

Turk et al. implemented hypergraph replication and repartitioning for social networks [31], which is one of the only approaches that uses the $(\lambda - 1)$ metric. Furthermore, they created a hypergraph model that utilizes temporal information in prior workloads to predict future query patterns. To reflect changes in the workload they divide the workload Q into $T = \{t_1, ..t_T\}$ time spans and weight each hyperedge in time span t with the *decay factor* $\alpha = \frac{|Q_t \cap Q_T|}{|Q_T|}$, where Q_t denotes the set of queries in time span t. This decay factor describes the similarity of time span t to the most current time span [31]. The motivation of weighting hyperedges with the decay factor is that workloads in social networks consist of requesting the latest tweets or news-feeds of friends, thus older tweets or news will probably not be queried again.

3.7. Approaches for OLAP Workloads

All approaches mentioned in the previous sections are designed for OLTP workloads. However, there are also methods for OLAP workloads. Because our work is designed for OLTP workloads, we just give a brief overview on them.

In contrast to the goal in OLTP settings which is to minimize all distributed transactions, in OLAP settings the goal is to co-locate the pair of tuples that participate in a join on the same machine, but distribute different pairs to gain parallelism. Figure 3.7 shows the difference between these approaches: In OLTP setting two tables participating in a join

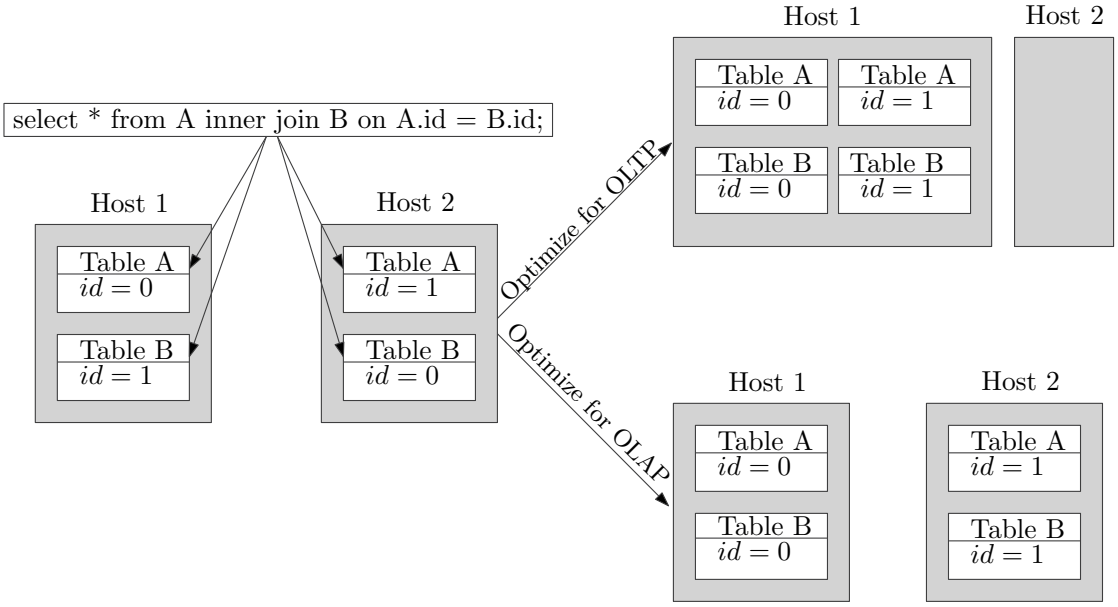


Figure 3.7: Shard placement optimization for OLTP and OLAP approaches ignoring any balance constraint

would be placed on the same host completely, whereas in OLAP the matching tuples are distributed in the cluster, so that each host can process the result locally.

Nam et al. introduced *GPT* [22], an approach based on graph partitioning for OLAP workloads, which outperforms other state-of-the-art methods in terms of both storage overhead and query performance. They create a *join graph* where the vertices represent tables, an edge indicates the join predicates of the connected tables and the weight of the edges describes the frequency of the joins. This graph is then used to derive the partitioning scheme that minimizes the joins that are executed in distributed machines.

4. Integrating a Workload-Aware Reassignment Framework into SAP Vora

To optimize the shard assignment in Vora, we implement a workload-aware reassignment framework that reduces the number of distributed queries while keeping the load on the hosts balanced. We model the workload as a hypergraph and use the state-of-the-art hypergraph partitioner KaHyPar to assign shards to hosts.

To the best of our knowledge, this is the first time that such an approach is implemented in a commercial enterprise database system. Previous approaches implemented their solutions in experimental databases like *H-Store* or *Relational Cloud* [8, 25, 26] that are designed for research purposes. In contrast to Vora, such database systems are much less complex because they lack failure resistance, failure recovery, authentication, auditing or support of multiple engine types. Thus, this thesis implements and examines the effects of optimizing the shards assignment via hypergraph partitioning in a complex enterprise database system.

The fist section of this chapter describes our approach and the second section gives a detailed overview on the implementation in Vora.

4.1. Approach

In Vora tables are horizontally partitioned into shards $S = \{s_1, ..., s_n\}$ using hash or range partition functions and assigned to hosts in the cluster in round-robin fashion. However, distributed transactions are expensive in *OLTP* settings [8] and should be kept to a minimum. On the one hand they lead to communication overhead between the involved hosts and on the other hand they increase the load of the system because each involved host has to execute the query. Our approach tries to optimize the shard assignment to hosts while keeping the load across the hosts balanced.

First of all, our approach is workload-aware, thus it monitors the executed queries $Q = (q_1, ..., q_m)$ where each query $q \in Q$ touches a given set of shards. Formally, we define for all queries $q \in Q$ that $q \subseteq S$. We construct the workload hypergraph as follows: $H = (S, E = Q, c, \omega)$, which depending on the weight functions can be the same workload model as *SWORD* uses [26]. Note that we assign the query sequence Q to hyperedge set E, which consists of all distinct queries of the workload (in general $|E| \leq |Q|$). The functions c and ω are chosen according to a preconfigured *weight policy*.

One weight policy is the *frequency weight policy*:

$$\forall s \in S : c(s) = |\{q \in Q \mid s \in q\}| \tag{4.1}$$

$$\forall e \in E : \omega(e) = |\{q \in Q \mid e = q\}|, \tag{4.2}$$

where $c(s)$ denotes the number of queries which shard s is part of and $\omega(e)$ denotes the number of queries that touch the same shards as hyperedge e. This workload hypergraph is the same model as *SWORD* uses in their approach.

By partitioning the workload hypergraph H using the frequency weight policy, we place frequently co-accessed shards together, thus minimizing the number of distributed queries and balance the load so that all hosts have to process a nearly equal number of queries.

Another policy we examine is the *execution time policy* for which we define the functions $t_{exec}(q)$ that returns the execution time for a query and $t_{exec}(q, s)$ that returns the execution time of a query q on the host where shard s is placed on. Using this functions we can define $c(s)$ and $\omega(e)$ as:

$$\forall s \in S : c(s) = \sum_{\substack{q \in Q \\ s \in q}} t_{exec}(q, s) \tag{4.3}$$

$$\forall e \in E : \omega(e) = \sum_{\substack{q \in Q \\ e = q}} t_{exec}(q)\,, \tag{4.4}$$

where $c(s)$ denotes the total execution time of all queries where s is part of and $\omega(e)$ denotes the total execution time of queries that touch same the shards as hyperedge e.

By partitioning the workload hypergraph H using the execution time weight policy, we place shards together that are frequently co-accessed by long running queries (e.g., multi table joins) and balance the load so that all hosts have a nearly equal execution time.

We model the workload on shard level instead of tuple level because Vora is only able to track access on shard level. However, shards are an aggregation of tuples and using such aggregations instead of working on tuple level showed greater scalability and less sensitivity to workload changes [26].

4.2. Implementation

To trigger the reassignment we implemented a dedicated SQL command in Vora, which is shown in Listing 4.1. Our approach then creates the hypergraph model based on the monitored queries Q, partitions it using KaHyPar, and uses its result to move the shards. To be able to partition the hypergraph using KaHyPar, we integrated it as a library into Vora.

```
01 | REASSIGN PARTITIONS WITH 'HYPERGRAPH' OPTIONS (
02 | EPSILON 0.1, WEIGHT_POLICY 'FREQUENCY',
03 | TRANSFORMATION 'SCHISM', OBJECTIVE 'KM1', SAMPLING_FACTOR 0.5
04 | );
```

Listing 4.1: Reassignment query syntax example

All parameters of the reassignment statement are configurable. Their purpose and the possible values are shown in Table 4.1.

Most of our work is implemented in the landscape component of Vora because it is responsible for data placement (see Section 2.2.2 for the architecture overview of Vora). In

`WITH`	The method that is used to optimize the assignment. It can be either `SWORD`, `CLAY` or `KaDaRea` (described in Chapter 5).
`EPSILON`	The imbalance parameter ε passed to the hypergraph partitioner.
`WEIGHT_POLICY`	The weight policy that determines the weights in the hypergraph. Possible values are `frequency` or `execution_time`.
`TRANSFORMATION`	The transformation that is used to enrich or modify the workload model. It can be either `schism` to transform it into Schism's workload model or `none` to keep the unmodified model.
`OBJECTIVE`	The objective that is minimized by the hypergraph partitioner. It can be `km1` or `cut`.
`SAMPLING_FACTOR`	Indicating the fraction of queries that are sampled before building the workload model. It can be any numeric value in $(0, 1]$

Table 4.1: Description of parameters for reassignment query

this component we aggregate statistics about each executed query, which are extracted and forwarded from transaction coordinator to landscape manager. These statistics contain information about touched shards of a query and the execution times on each host, which are used to build the workload hypergraph.

Figure 4.1 shows the design of our approach and the steps that happen if a reassignment is triggered. The first step is that the landscape server builds an internal representation of the workload hypergraph. For each query $q \in Q$ that is stored in a component which we call the statistic collector, the spanned shards are added as vertices to the hypergraph and a hyperedge connecting these shards is added.

If the *sampling factor* is set, we sample the queries before creating the workload hypergraph. Thus, we are able to simulate the effects of having smaller monitoring timespans because in some systems monitoring each query could be very expensive. Moreover, the landscape component scans for shards that are not touched by any query and adds them to hypergraph to receive the complete unweighted workload hypergraph.

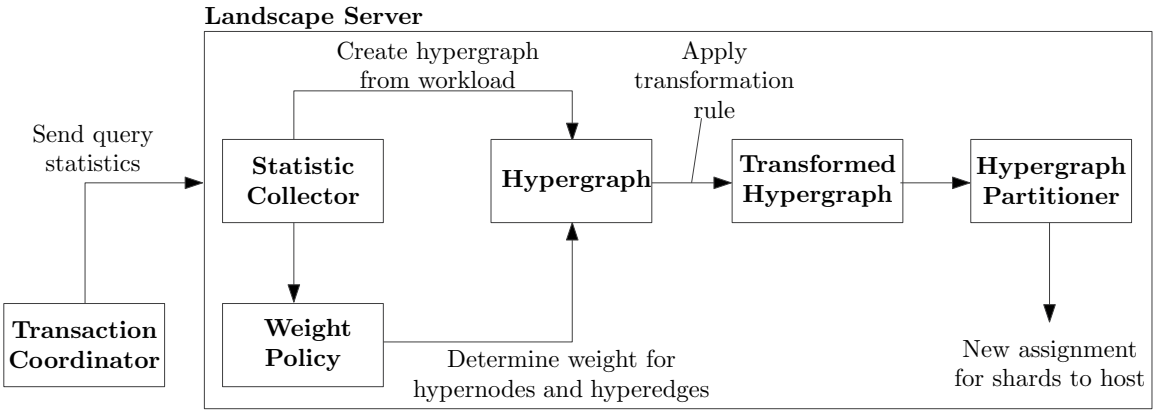


Figure 4.1: Design of shard reassignment in Vora

The workload hypergraph is transformed into a weighted hypergraph by applying the weight policy specified in the repartitioning statement presented in Section 4.1.

Afterwards, the hypergraph model is transformed based on a *transformation rule*. Such a rule can be used to transform our hypergraph model into Schism's graph model by applying clique expansion.

Both the weight policy and the transformation rule are easily extendable to experiment with other weight policies or to extend the basic model with more advanced techniques.

Finally, the model is transformed into the input data structure of KaHyPar's interface and it is then called to partition the hypergraph into k (number of hosts) blocks with the configured imbalance parameter (specified by option `EPSILON`) and objective function (specified by option `OBJECTIVE`). KaHyPar is integrated into Vora and the communication between Vora and KaHyPar takes place by using its C style library interface.

The result of the hypergraph partitioning are k disjoint blocks that describe which shards should be placed together. However, before moving the shards to their new hosts, we want to find a mapping between the current assignment $\Psi = \{\Psi_1, ..., \Psi_k\}$ and the new one $\Pi = \{\Pi_1, ..., \Pi_k\}$, such that the required moves are minimized.

To solve this problem we create a weighted bipartite graph $G = (V = L \cup R, E, \omega)$ (see Definition 2.3) where the nodes on the left side L represent the blocks of the new assignment, and the nodes on the right side R represent the blocks of the current assignment. An edge between a node $u \in L$ and $v \in R$ is weighted with the number of vertices which the corresponding blocks Π_u and Ψ_v have in common. More formally, $\omega(u, v) = |\Pi_u \cap \Psi_v|$. In order to minimize the number of moves, we try to find a permutation of the new assignment $\bar{\Pi}$, such that $\sum_{i \in \{1,...,k\}} |\Psi_i \cap \bar{\Pi}_i|$ is maximized. This is an instance of the maximum weighted bipartite matching problem (see Definition 2.5) and can be solved by applying the Hungarian algorithm (see Section 2.1.4) on the previously created bipartite graph G [19].

Figure 4.2 shows the effect of our algorithm: Initially the shards A_1, B_1, and C_1 are placed on host h_1. Partitioning the workload hypergraph reduces the distributed queries by placing A_1, A_2, and B_2 together, however the partitioning result would place them on h_1, resulting in a total of 4 moves. By finding the maximum weighted bipartite matching between the new assignment and the current assignment, A_1, A_2, and B_2 are placed on h_2, reducing the required moves to 2.

At the end, the landscape component sends the move instructions to the transaction coordinator and updates the shard information in the catalog server, so that incoming queries are routed to the new hosts of the shards.

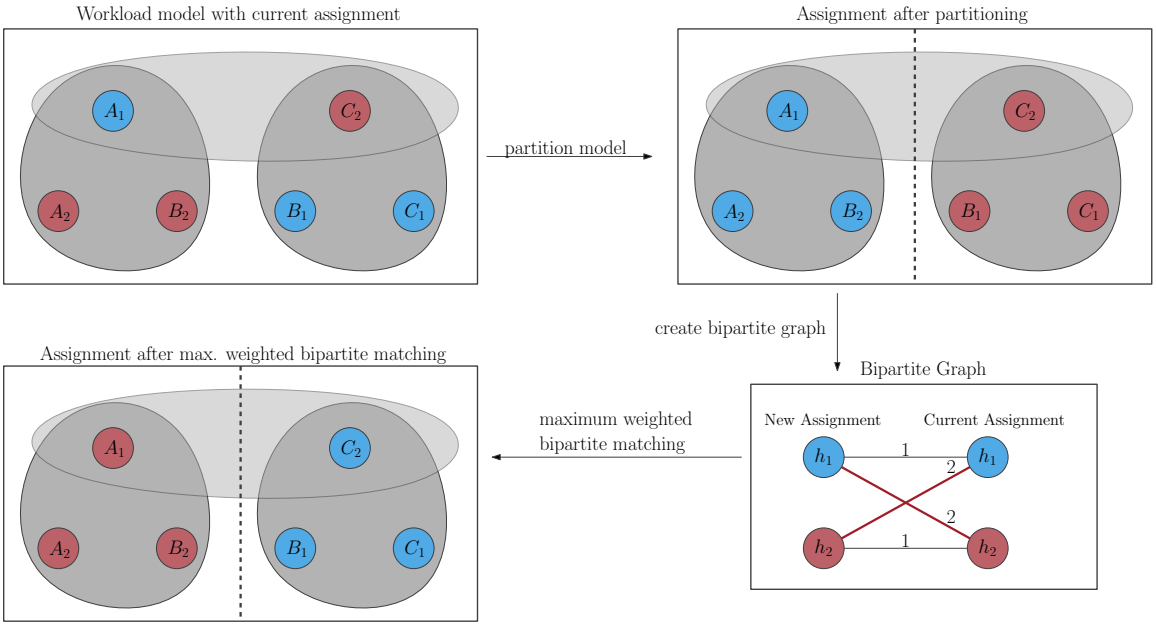


Figure 4.2: Reduced number of moves by using maximum weighted bipartite matching. Color indicates host of shard.

5. KaDaRea - A Peak- and Pattern-Aware Database Reassignment Technique

In this chapter we propose *Karlsruhe Database Reassigning*, called *KaDaRea*, a novel approach for solving the allocation problem in context of shard placement. Our technique divides the workload into time intervals and partitions the workload hypergraph for each of it. With a special rating function we are able to optimize shard placement in presence of workload peaks or even changes in the workload patterns, which would lead to several shortcomings with the traditional approach, such as assignments that are not optimized for peaks in the workload or assignments that are not balanced at any point in time.

Our approach is novel as to the best of our knowledge there is no other technique which partitions the workload hypergraph for each time interval. Turk et al. consider the time of execution for computing the hyperedge weights, but in contrast to our approach they create a single workload hypergraph and partition it [31]. Without partitioning the workload hypergraph for each time interval it is not possible to detect changes or peaks in the workload.

Firstly, we outline the main idea and the motivation of using such a technique, before we go into implementation details.

5.1. Motivation

In real world applications the processed load by a distributed database system varies heavily depending on the time of the day or external events. On the one hand, the intensity of the load changes, thus there are times where the load is low and peak times where the load is much higher. Most systems can handle the low or regular load without any advanced repartitioning techniques. However, they fail to deliver their service during peak times, leading to a huge dissatisfaction for the customers. Examples for this peak times are popular sport events leading to failures at sport streaming services or launches of new products at web shops.

On the other hand, the mixture of the load changes over time leading to patterns in the workload. An example for this are multi-tenant distributed databases, where teams from different regions are working on the same database. To illustrate, there are two teams, one in Central Europe and one in the US, which access different tables of the database to fulfill their tasks. Because the US workday is much later than the one in Central Europe, it leads to two different patterns of workloads, one for the team in Central Europe and one for the team in the US. Partitioning the workload model of these two workload patterns without considering when the workload is executed could result in an assignment that is bad for

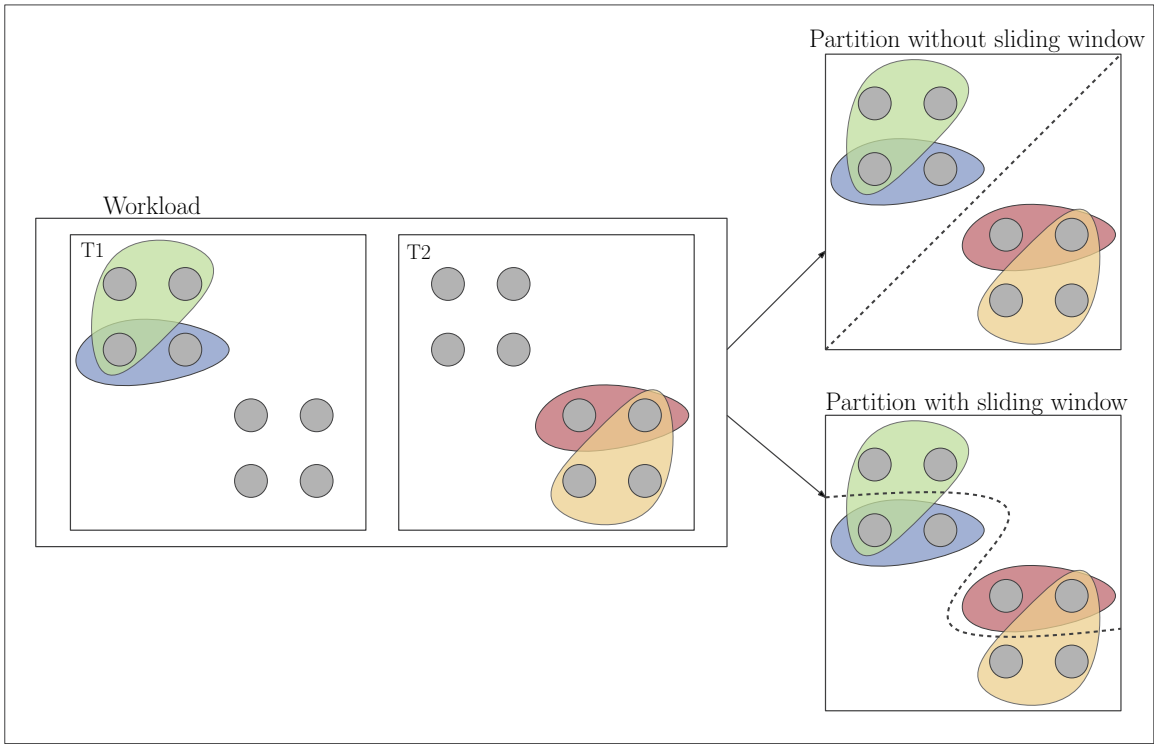


Figure 5.1: Using a sliding window results in a balanced assignment for changing workload patterns

both patterns because it is imbalanced at any point in time even if the partitioning result of the combined workload is balanced.

To solve these problems we propose an approach that is based on splitting the workload into several time slices and then partition each workload hypergraph independently, similar to a sliding window. The resulting assignments for the different time intervals are aggregated in a rating function and weighted with the relative load inside a time slice. At the end, a shard is assigned to the block with the highest rating.

The first advantage of this approach is that it gives windows with peak load more impact to be able to adapt better to peak times. Secondly, by partitioning the hypergraph using a sliding window the weight of nodes from different time slices do not interact when trying to find a balanced partitioning as they do when we partition the hypergraph without sliding windows. Figure 5.1 depicts how the assignment is different if a sliding window is used and the workload patterns are changing. In time interval $T1$ only the upper shards are queried and in $T2$ only the lower ones. If the resulting workload hypergraph is partitioned without considering when the queries are executed, the best assignment is to assign the upper shards to one host and the lower ones to the other host. However, this assignment is heavily imbalanced as in one time frame one host is busy while the other on is idling. A better assignment regarding performance and load balance is to cut the upper and lower group of shards and assign one part of the each group to a host, resulting in a utilization of both hosts at any time.

5.2. Implementation

In the following section we present our sliding window algorithm in detail, which was motivated and sketched in the section before.

For a query $q \in Q$ we define functions $t_s(q)$ and $t_e(q)$ with $t_s(q) < t_e(q)$ that return the start and end time of a query. Using this functions we can construct the sequence of all queries Q_w and the workload hypergraph H_w for time interval $w = [t_s, t_e)$:

$$Q_w = (q_i \in Q | t_s \leqslant t_s(q_i) \wedge t_e(q_i) < t_e) \tag{5.1}$$

$$H_w = (S, E = Q_w, c_w, \omega_w)\,. \tag{5.2}$$

The weight functions c_w and ω_w are one of the weight policies defined in Section 4.1 based on the queries in Q_w.

Let $w_{size} > 0$ be the sliding windows size and $w_{step} > 0$ be the sliding window step with $w_{step} \leqslant w_{size}$, we can define the set of windows $W = \{w_0, ..., w_l\}$, where

$$w_i = [t_{start} + i \cdot w_{step},\ t_{start} + i \cdot w_{step} + w_{size}] \text{ with } t_{start} = \min(t_s(Q))\,. \tag{5.3}$$

Parameter l is defined as the minimum index where for $w_l = [t_s, t_e)$ the condition $t_e > \max(t_e(Q))$ holds.

We can now calculate the partitioning of H using the sliding window algorithm described in Algorithm 1. For each window w we create the hypergraph H_w and partition it into k disjoint blocks $\Pi = \{\Pi_1, ..., \Pi_k\}$. For all shards s we store a rating to each block Π_i that reflects how often s was placed on Π_i multiplied with a window factor. The factor of a window gives windows with a higher load more impact, leading to an assignment that is optimized for workload peaks. It is defined as follows:

$$f(w_i) = \left(\frac{c_{w_i}(S)}{M}\right)^2 \text{ with } M = \max_{i \in [0,l]} (c_{w_i}(S))\,. \tag{5.4}$$

However, before increasing the rating of the blocks, the most similar permutation between the last partitioning result Π_{last} and the current result Π must be found because a partitioning result can be equal to the last partition but only differ in the indices of the blocks. This is the same instance of the maximum weighted bipartite matching problem that we solved in the previous chapter by applying the Hungarian algorithm (see Section 2.1.4). At the end we place each shard to the block with the highest rating.

We extended the reassignment query syntax to be able to configure the usage of KaDaRea and the parameters for it. See Listing 5.1 for a syntax example with the additional options to use KaDaRea and to configure w_{size} and w_{step}, whereby the unit for both are seconds.

```
01 | REASSIGN PARTITIONS WITH 'KaDaRea' OPTIONS (
02 |         EPSILON 0.1, WEIGHT_POLICY 'FREQUENCY',
03 |         SLIDING_WINDOW_SIZE 50, SLIDING_WINDOW_STEP 10
04 | );
```

Listing 5.1: Reassignment query syntax for sliding window

Algorithm 1: KaDaRea

```
  input  : Q, w_size, w_step, k
  output : Π
1 t = min(t_s(Q))
2 W = ()
3 do // determine all windows
4 |   W.append([t, t + w_size])
5 |   t = t + w_step
6 while t ⩽ max(t_e(Q))
7 ∀s ∈ S : ∀i ∈ {1, ..., k} : R(s, i) = 0          // set initial rating of shards to blocks
8 Π_last = ∅                                      // partitioning result of previous window
9 M = max_{w∈W}(c_w(S))
10 foreach w ∈ W do
11 |   H_w = (S, E = Q_w, c_w, ω_w)
12 |   Π = partition(H_w, k)
   |   // find most similar permutation to Π_last by max. weighted bipartite matching [19]
13 |   Π = maximumWeightedBipartiteMatching(Π, Π_last)
14 |   foreach s in S do
15 |   |   R(s, i) += c_w(s) / M where s ∈ Π_i
16 |   Π_last = Π
17 foreach s ∈ S do
18 |   Assign s to Π_i where i = arg max_{i∈[1,k]}(R(s, i))
```

This approach has several advantages: The first one is that queries during a peak load have more impact on the partitioning result. Another benefit is that if the workload patterns change over time, partitioning the hypergraph using a sliding window will lead to a more balanced solution, which results in a better utilization of the system.

6. Evaluation

In this chapter we compare the performance of our workload aware reassignment framework to the current state in Vora of assigning the shards in round robin fashion. Additionally, we compare our approach to other state-of-the-art techniques and evaluate KaDaRea.

6.1. Experimental Setup

For the evaluation of our work we deployed Vora with *Google Kubernetes Engine* on a *GKE* cluster. Each node in the cluster has 8 virtual CPU cores and 30 GB of main memory. To avoid network effects negatively affecting the benchmark results like slow routers or corporate firewalls, we execute the benchmarks inside the cluster on a separate node instead of executing them on a client machine outside the cluster.

We evaluate our approach by running the widely used *TPC-C*[1] and *TPC-E*[2] benchmarks, which are implemented as Java applications and connect to the database via a JDBC connection. Our benchmarks are based on open source implementations and we adapted them to work with Vora. These benchmarks are described in Section 6.1.2 and Section 6.1.3 in more detail.

An *experimental run* consists of an initial benchmark run, where all queries are monitored. Afterwards, the reassignment statement is triggered and based on the configured reassignment strategy the shard landscape is optimized. Finally, we run the benchmark a second time on the optimized assignment to compare its results with the results of the initial benchmark run.

We partition the tables in both benchmarks by their primary key using a hash partition function, leading to k shards for each table, where k is the number of hosts (see Section 2.2.1 for database partitioning). Furthermore, we create a sufficient amount of client connections to Vora that query the system in parallel such that it is fully utilized. To avoid side effects between multiple experimental runs, all hosts are restarted after each run.

6.1.1. Methodology

There are two main metrics we examine in this thesis. The first one is the speed-up in throughput S which describes how much the throughput N_{after} after the optimization increased compared with the throughput N_{before} of the benchmark execution before the

[1] `https://github.com/AgilData/tpcc`

[2] `https://github.com/apavlo/h-store/tree/master/src/benchmarks/edu/brown/benchmark/tpce/`

optimization, formally defined as:

$$S = \frac{N_{after}}{N_{before}} . \tag{6.1}$$

The second one is the share of distributed queries weighted with the number of spanned hosts after the optimization took place. Hence, queries which span many hosts lead to a bigger value than queries which only span a few. D is formally defines as:

$$D = \frac{\sum\limits_{q \in Q \,:\, \lambda(q,\Pi)>1} \lambda(q, \Pi)}{\sum\limits_{q \in Q} \lambda(q, \Pi)} . \tag{6.2}$$

We are not able to make absolute throughput or response time numbers public due non-disclosure clauses, thus we transform the throughput of the benchmarks and response time of the transactions into normalized values by dividing each value with the maximum.

To visualize the shard assignment we create *workload graphs* such as the one depicted in Figure 6.4. This workload graph is created by transforming the workload hypergraph using the bipartite transformation. The large nodes in the figure represent shards where the node label represents the table of the shard and the node color represents the host of the shard. Shards are connected to queries, represented by small nodes in the figure. Thus, shards that are connected to the same node are part of the same query. The thickness of a edge represents the frequency of a query.

Furthermore, to visualize the throughput over time we output the throughput of the last 20 seconds during the benchmark execution. Using this information, *throughput over time plots* are created like the one shown in Figure 6.10. The depicted throughput is the normalized throughput which is computed by dividing each value with the maximum measured throughput in the experiment. Additionally, the figure shows the 0.95 confidence interval as a band around the curve.

6.1.2. TPC-C

TPC-C [21] is a classical OLTP benchmark created in 1992 that models a warehouse with its customers using 9 tables, 5 different transaction types, and a data size of about 500.000 records per warehouse. We configured TPC-C to consist of a single warehouse. A transaction type consists of several queries. For example, such a transaction type is the *new order transaction* that creates a new order and updates relevant stock entries. The database schema of TPC-C is depicted in Figure 6.1.

At the start of the benchmark a number of preconfigured clients are created, which query the database in parallel. Each client randomly chooses a transaction type from a predefined probability distribution and its parameters and executes all queries related to the transaction. This is repeated until the benchmark ends.

TPC-C is a rather simple benchmark compared to TPC-E, however it is still widely used to evaluate the results of partitioning approaches [3, 6].

The measured metric in TPC-C is the number of completed new order transactions per minute.

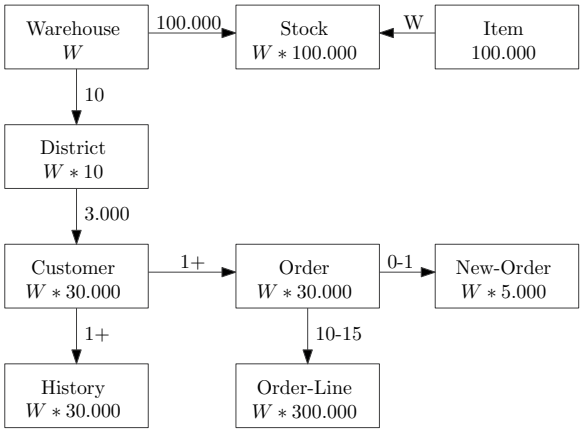


Figure 6.1: TPC-C database schema (adapted from [30])

6.1.3. TPC-E

TPC-E [6] is a more recent OLTP benchmark released in 2007 that models a brokerage house with customers, stocks and trades using 33 Tables and 10 different transaction types. We configured the data size of TPC-E to be about 10 GB (~177.500.000 records). The customer and stock exchange are modeled to drive transactions at the brokerage house, which is the system under test, as shown in Figure 6.2. The brokerage house manages information for the customers, the brokers and the stock market. Therefore, customer query the brokerage house to place orders or get information about the market. The brokerage house then updates its database or sends a request to the stock exchange. Also, the stock exchange sends market ticker feeds to the brokerage house which then reacts to these feeds by submitting orders if certain conditions in the market are met. The measured metric in TPC-E is the number of all completed transactions per seconds.

TPC-E is designed to be a more realistic benchmark and is far more complex than TPC-C, as it incorporates realistic data skews and multi table joins [6]. The database schema of TPC-E is depicted in Figure A.1.

The flow of the benchmark is the same as the flow of TPC-C: At the start a number of clients are created which randomly choose a transaction type with its parameters and execute all queries related to the transaction. This is repeated until the benchmark ends.

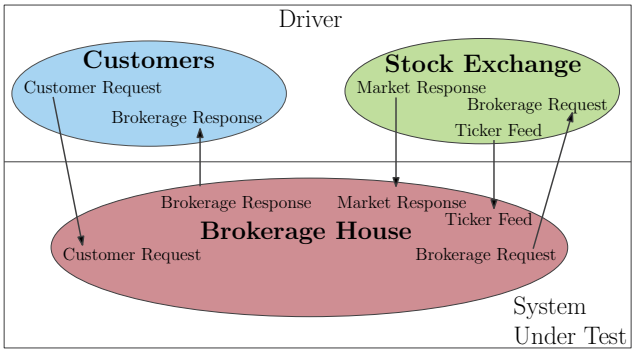


Figure 6.2: TPC-E models a financial brokerage house (adapted from [6])

6.2. Parameter-Tuning Experiments

In this section we examine the effects of different parameters on the reassignment quality in order to find a proper setting to compare our results with the current state in Vora and state-of-the-art approaches.

If not stated otherwise, we run the benchmarks on 4 hosts, using the connectivity objective, the frequency weight policy, an imbalance parameter ε of 0.1, a benchmark measuring time of 300 seconds, and warm up time of 30 seconds per benchmark execution.

6.2.1. Imbalance

Figure 6.3 shows the speed-up in throughput S and the share of distributed queries D weighted with the number of spanned hosts for varying ε parameter values.

As one can see, the throughput of the system increases greatly even for small ε values like 0.03. Also, D drops with increasing ε until it reaches 0, which means that there are no distributed queries in the benchmark execution after the optimization took place.

However, the throughput decreases at some point, hence the best performance cannot be reached by increasing ε until all distributed queries are eliminated. This happens because for big ε some hosts are idling while others are overloaded.

In fact, when looking at the TPC-E benchmark S drops significantly for $\varepsilon > 1$ for 4 and 8 hosts. For 2 hosts, there is just a small decrease for $\varepsilon > 0.1$, because in this case all distributed transactions can be eliminated using a rather small ε.

The optimal ε value differs depending on the number of hosts and increases as the number of hosts increases. Thus, depending on the number of hosts the proper ε parameter value must be found to maximize the throughput. We choose $\varepsilon = 0.1$ for 2 hosts, $\varepsilon = 0.25$ for 4 hosts, and $\varepsilon = 0.5$ for 8 hosts.

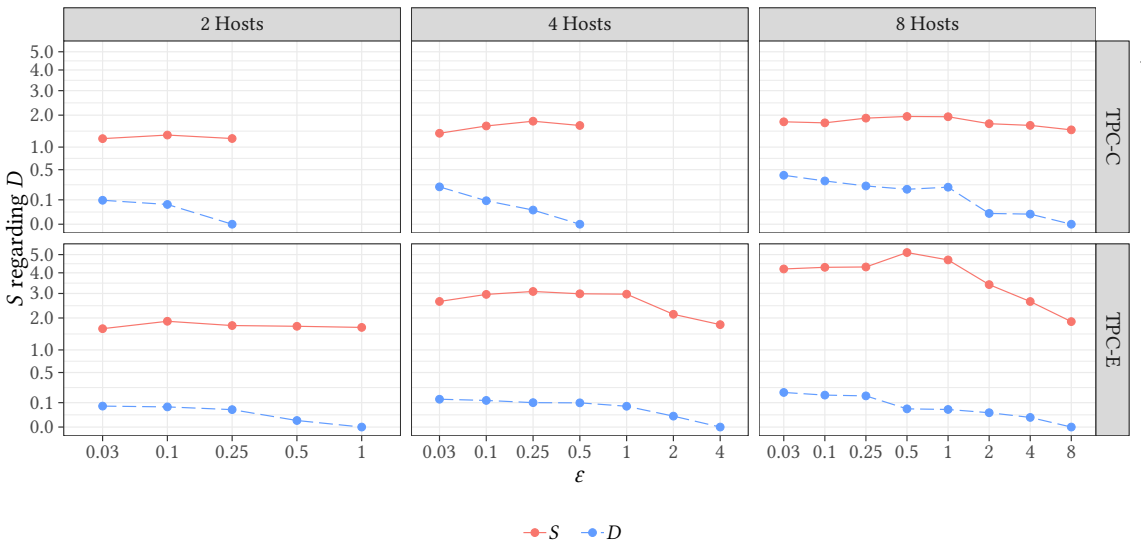


Figure 6.3: Evaluation of varying ε values regarding speed-up in throughput S and weighted share of distributed queries D

Table 6.1: Evaluation of cut and connectivity ($\lambda - 1$) metric regarding throughput in speed-up S and weighted share of distributed queries D

	4 Hosts				8 Hosts			
	TPC-C		TPC-E		TPC-C		TPC-E	
Objective	S	D	S	D	S	D	S	D
Cut	1.60	0.09	2.95	0.11	1.65	0.34	3.92	0.20
$\lambda - 1$	1.61	0.09	2.94	0.11	1.73	0.31	4.30	0.17

6.2.2. Objective Metric

This section examines the effects of using the cut or the connectivity ($\lambda - 1$) metric when partitioning the hypergraph. Some of the state-of-the-art techniques use the connectivity metric [4, 31], others use the cut metric [26, 34], however none of them compare the metrics.

If a hyperedge is cut and this results in a query that is distributed across 3 hosts, the performance is worse than if a hyperedge is cut and the query is distributed across 2 hosts because there is more communication overhead and one more host that has to process the query. This property is considered by the connectivity metric as it considers to how many blocks a hyperedge is connected to.

As Table 6.1 shows, there is no significant difference measurable for 4 hosts, but for 8 hosts the connectivity metric results in a larger speed-up in throughput S and less distributed queries D.

Thus, we use the connectivity metric in our experiments.

6.2.3. Weight Policy

In Section 4.1 we provided two weight policies: the frequency and the execution time policy. Nearly all state-of-the-art techniques model the weights of the hyperedges and vertices using the frequency policy, but none of them apply a weight function that considers the execution time of the queries.

We implemented the execution time policy in two variants, one where the execution time is used to determine both the vertex and hyperedge weights (Equation (4.3) and Equation (4.4)) and one where the execution time is only used to determine the hyperedge weight and the frequency is used for the vertex weights (Equation (4.1) and Equation (4.4)).

Table 6.2 shows that the frequency policy performs best, leading to the highest speed-up and the least distributed queries as both execution time polices perform significantly worse than the frequency policy for the TPC-E benchmark.

We observed that if the execution time policy for vertices and hyperedges is used, some shards are getting too heavy, resulting in hosts with only a few shards on them.

Furthermore, in the second case where the execution time is only used to determine the weight of the hyperedges, the performance is still worse than using the frequency policy

Table 6.2: Evaluation of frequency and execution time policies regarding speed-up in throughput S and weighted share of distributed queries D on 4 hosts

	TPC-C		TPC-E	
Weight Policy	S	D	S	D
Frequency	1.61	0.09	2.94	0.11
Execution Time (Vertices & Edges)	1.09	0.20	0.98	0.49
Execution Time (Edges)	1.60	0.10	1.28	0.40

for the TPC-E benchmark. This means that our assumption that reducing distributed long running queries would lead to a larger speed-up is wrong, instead the frequency of queries has more impact on the overall throughput.

Thus, the frequency weight policy should be preferred.

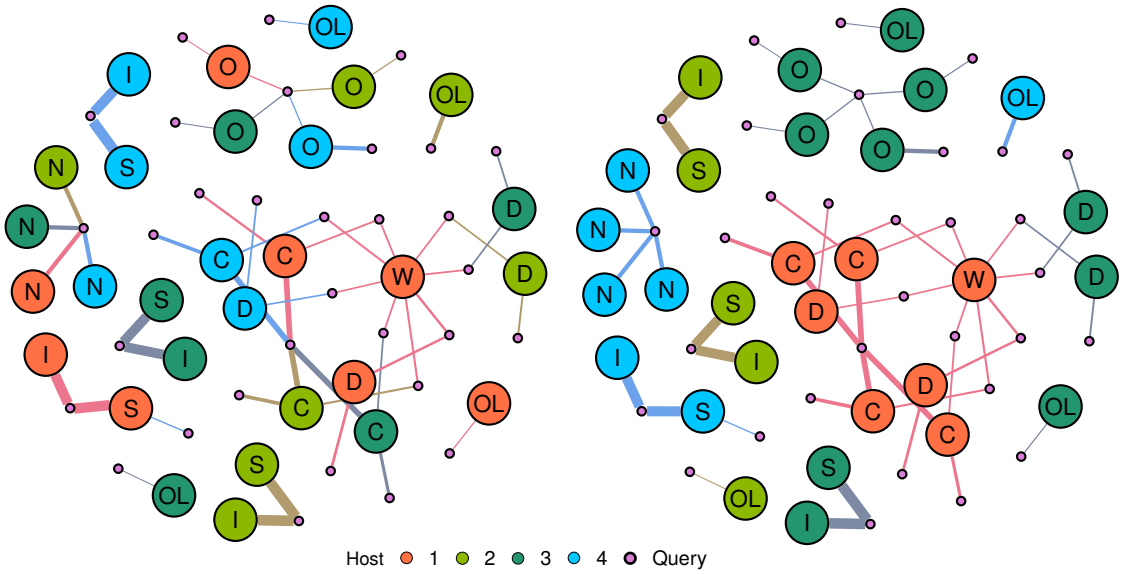
6.2.4. Sampling

In this section we evaluate the effect of sampling the monitored queries. As Table 6.3 shows, for both benchmarks 1% of the queries are enough to create a solid workload model that increases the throughput by a great extent. In the case of the TPC-C benchmark 10% of the queries are enough to create the best workload model for our approach. In contrast to TPC-C, we need 50% of all queries to create the best workload model for TPC-E.

Thus, in general it is sufficient to monitor only a fraction of the queries to get a good workload model.

Table 6.3: Evaluation of robustness to sampling regarding speed-up in throughput S and weighted share of distributed queries D on 4 hosts

	TPC-C		TPC-E	
Sampling Factor	S	D	S	D
1%	1.50	0.16	2.34	0.18
10%	1.61	0.08	2.58	0.15
25%	1.60	0.10	2.73	0.12
50%	1.62	0.08	2.94	0.10
100%	1.61	0.09	2.94	0.11



(a) Workload graph before reassignment (b) Workload graph after reassignment

Figure 6.4: Workload graph for TPC-C benchmark running on 4 Hosts. The node labels symbolize the table of the shard.

6.3. Comparison with Current State in Vora

This section evaluates the impact of our new reassignment framework compared to the current state in Vora of assigning the shards in round robin fashion in detail using the TPC-C and TPC-E benchmark.

The experiments in this section are executed with the following parameters that have been determined in the previous section: We use the connectivity metric, the frequency weight policy, a sampling parameter of 100% and an imbalance parameter $\varepsilon = 0.1$ for 2 hosts, $\varepsilon = 0.25$ for 4 hosts, and $\varepsilon = 0.5$ for 8 hosts.

6.3.1. TPC-C

A visualization of the shard assignment before and after the reassignment can be seen in Figure 6.4, which shows that after the reassignment most of the shards that are queried together are placed on the same host.

This leads to a big performance improvement depicted in Figure 6.5 which shows the normalized throughput over the running time of the benchmark before and after the reassignment. The throughput is much higher after the reassignment, leading to an average speed-up in throughput $S = 1.33$ for 2 hosts, $S = 1.78$ for 4 hosts, and $S = 1.94$ for 8 hosts.

Also, the distributed queries are reduced and all highly distributed queries, which are queries that access all hosts of cluster, are eliminated (see Figure A.2): The weighted share

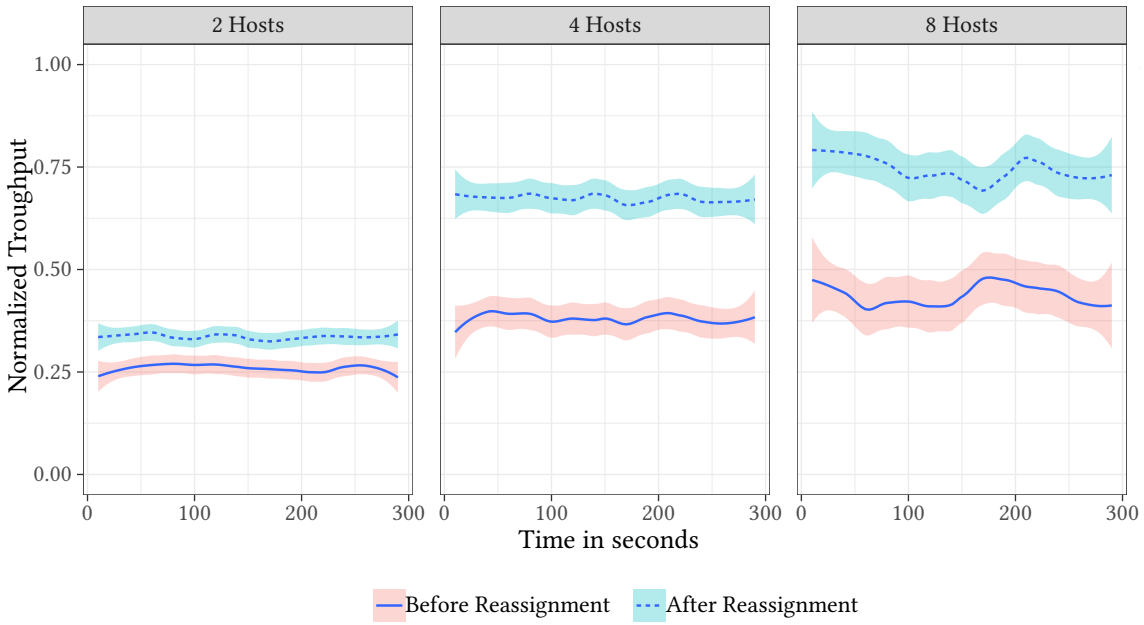


Figure 6.5: Normalized throughput (current/max) with 0.95 confidence interval before and after reassignment for TPC-C

of distributed queries D is reduced by 78 % for 2 Hosts, 92% for 4 Hosts, and 65% for 8 Hosts.

Figure 6.6 presents how the response time of transactions in TPC-C change by optimizing the assignment. Overall, the response time improves greatly and the variance for the transaction types decreases. The more distributed queries like joins or table scans a transaction contains, the greater is the speed-up of the response time. When looking at the implementation of the transactions, one can see that `Delivery` has 66% distributed queries whereas `OrderStat` has only 20% distributed queries, thus `Delivery` has a much larger speed-up. `NewOrder` does not improve much because most of its queries access table `Inventory(I)` and `Stock (S)` for which the assignment before the optimization was already good. The response time of `StockLevel` even decreases because it accesses single shard of table `District (D)` and `OrderLine (OL)`. This means that its queries have not been distributed before the reassignment, but the shards of the tables have been placed on different hosts, thus simultaneous queries to `District (D)` and `OrderLine (OL)` could be parallelized to a greater extend.

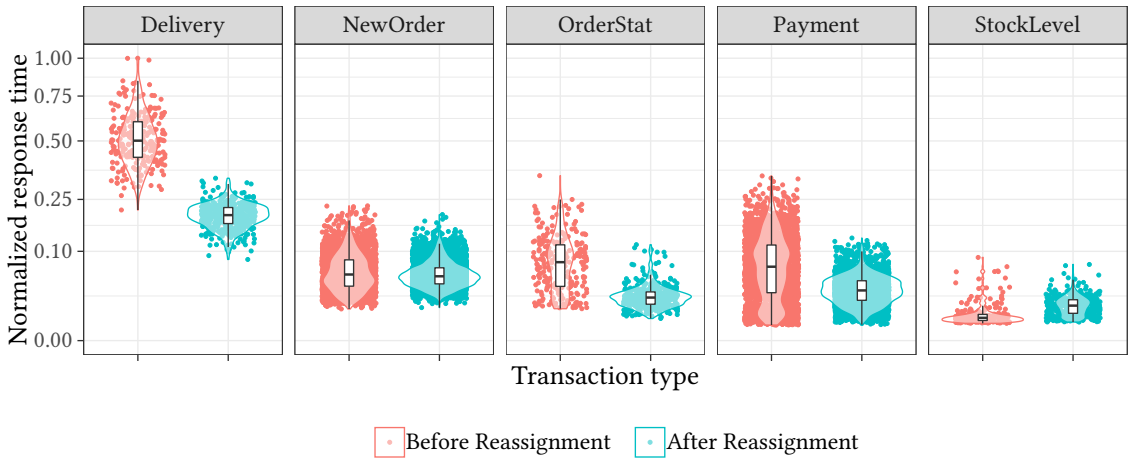


Figure 6.6: Normalized response times (current/max) for each transaction type of TPC-C on 4 hosts

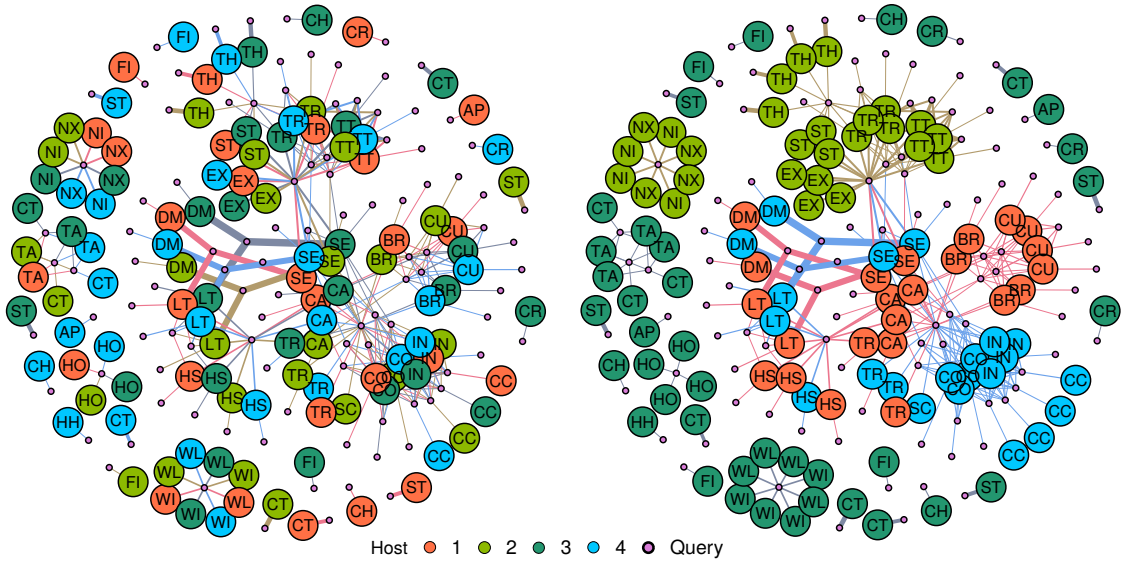
6.3.2. TPC-E

As already mentioned in Section 6.1 the TPC-E benchmark is much more complex than the TPC-C benchmark. This can be seen in Figure 6.7 which depicts the shard assignment before and after the reassignment and shows that shards that are frequently queried together are placed on the same host after the optimization.

The performance improvement is even higher than for the TPC-C benchmark: The speed-up in throughput is $S = 1.88$ for 2 hosts, $S = 3.09$ for 4 hosts, and $S = 5.11$ for 8 hosts. One important thing to note is that the throughput before the reassignment does not increase if hosts are added, as Figure 6.8 shows. This means that one cannot scale the throughput of the system by simply adding hosts, one must also optimize the shard assignment to get any performance improvements.

Also, the distributed queries are reduced and again all highly distributed queries are eliminated (see Figure A.3): The weighted share of distributed queries D is reduced by 65% for 2 Hosts, 79% for 4 Hosts, and 91% for 8 Hosts.

Figure 6.9 presents how the response time of transactions in TPC-E change by optimizing the assignment. Overall, the response time improves greatly and the variance of the response time for a transaction type decreases. `BrokerVolume` shows the greatest speed-up of response time because it contains a join across 7 tables whereas `TradeLookup` does not improve much because it contains nearly no distributed queries. `MarketWatch` consists of mostly joins, however it does not improve. That is the case because the initial round robin assignment places the partitions in such a way that the joins are non distributed thus we cannot improve the assignment for this transaction any more.



(a) Workload graph before reassignment (b) Workload graph after reassignment

Figure 6.7: Workload graph for TPC-E benchmark running on 4 Hosts. The node labels symbolize the table of the shard.

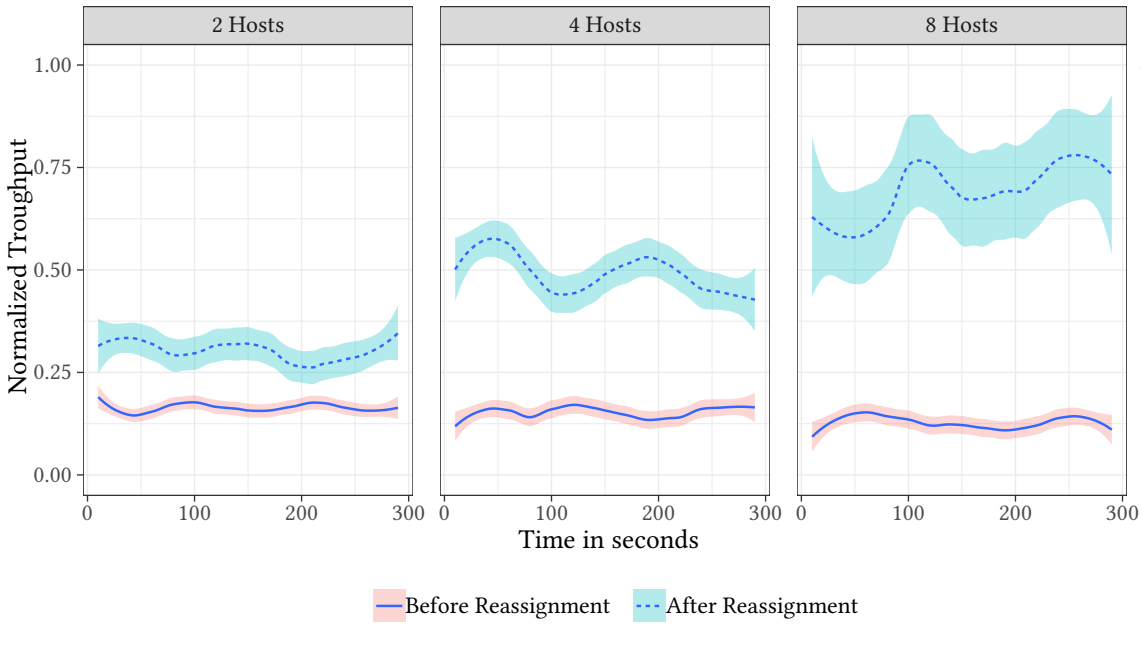


Figure 6.8: Normalized throughput (current/max) with 0.95 confidence interval before and after reassignment for TPC-E

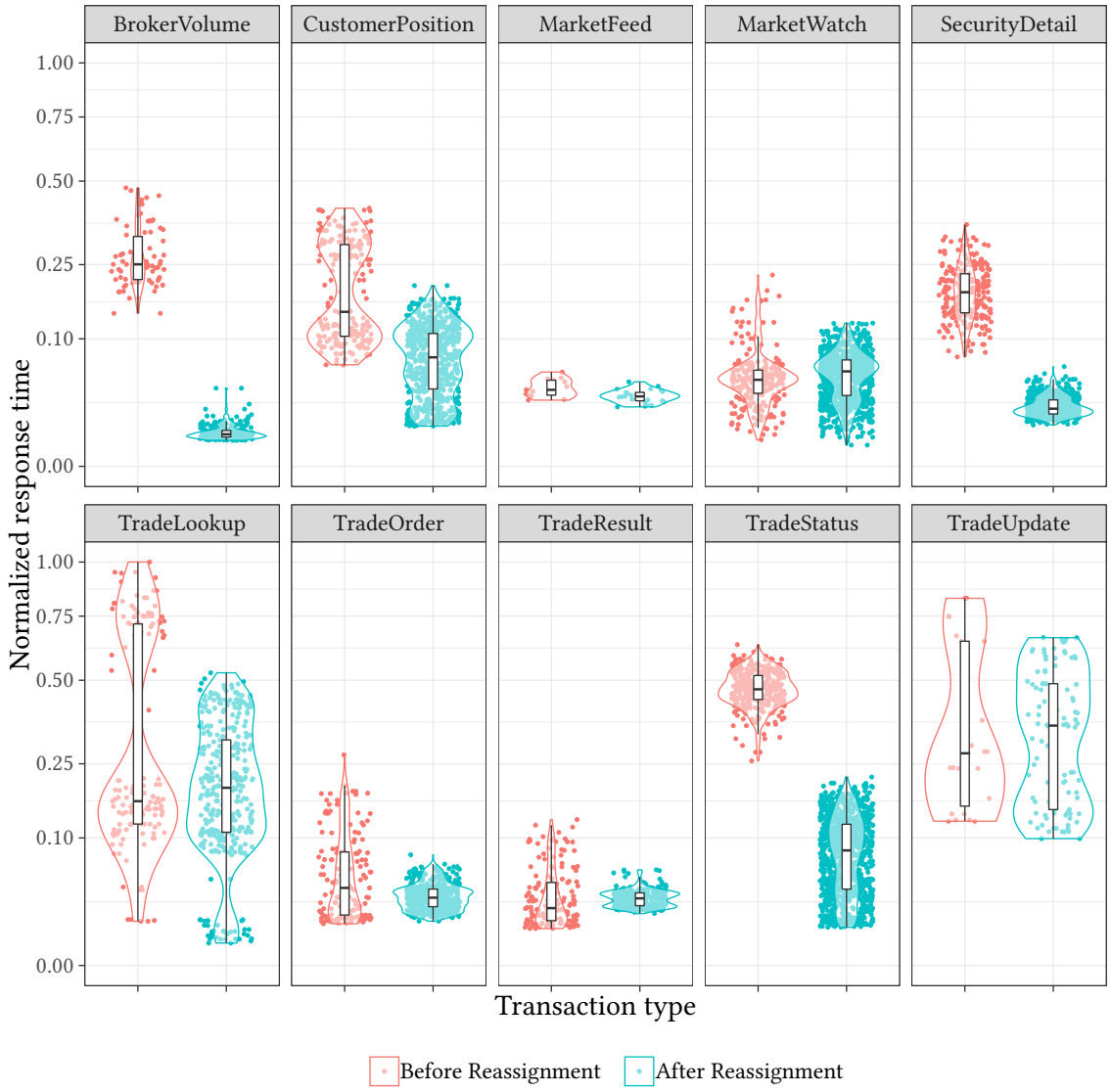


Figure 6.9: Normalized response times (current/max) for each transaction type of TPC-E on 4 hosts

6.4. Comparison with State-of-the-Art Approaches

This section evaluates our approach in comparison to the state-of-the-art approaches Schism and Clay. As already mentioned in Section 4.1, SWORD uses the same workload model as our approach, therefore we refer to our approach as SWORD in this section.

The experimental parameters are the same as in Section 6.3.

6.4.1. TPC-C

Figure 6.10 shows that the throughput of Schism is as high as the throughput of SWORD. This is the case, because TPC-C consists mostly of simple queries that can be represented well in a graph model, thus Schism and SWORD lead to the same shard assignment, depicted in Figure 6.11. However, Clays result is not balanced because there are too many shards placed on host 2. Furthermore, it stops even if there are still many distributed queries, thus the performance is much worse compared to SWORD resulting in a speed-up in throughput $S = 1.05$ whereas Schism and SWORD accomplish a speed-up in throughput $S = 1.74$.

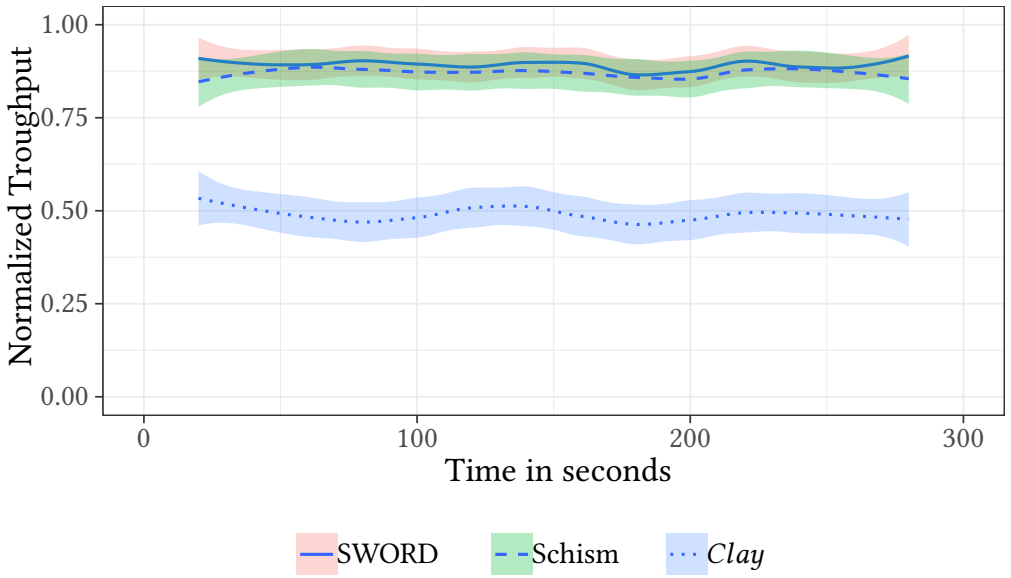
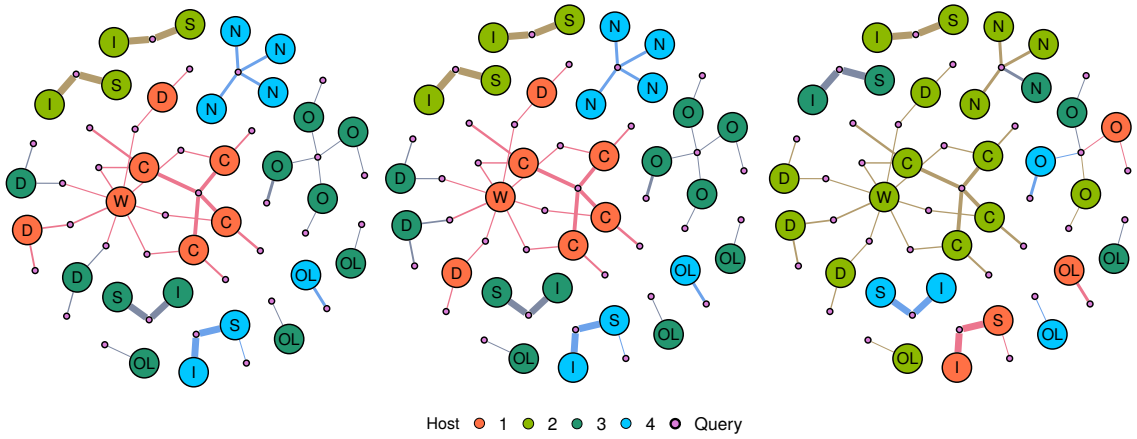


Figure 6.10: Comparison of SWORD, Schism, and Clay: Normalized throughput (current/max) with 0.95 confidence interval after reassignment for TPC-C on 4 hosts



(a) SWORD's workload graph (b) Schism's workload graph (c) Clay's workload graph

Figure 6.11: Comparison of SWORD, Schism, and Clay: Workload graphs after assignment optimization for TPC-C benchmark running on 4 Hosts. The node labels symbolize the table of the shard.

6.4.2. TPC-E

While Schism created a good assignment for the TPC-C benchmark, it fails to create a good assignment for the more complex TPC-E benchmark. As Figure 6.12 shows, using

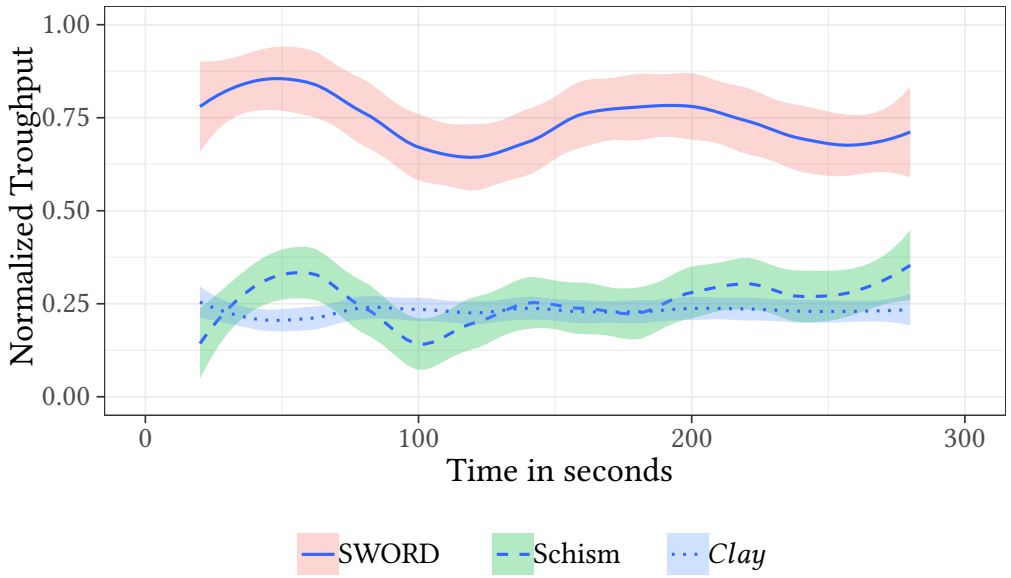
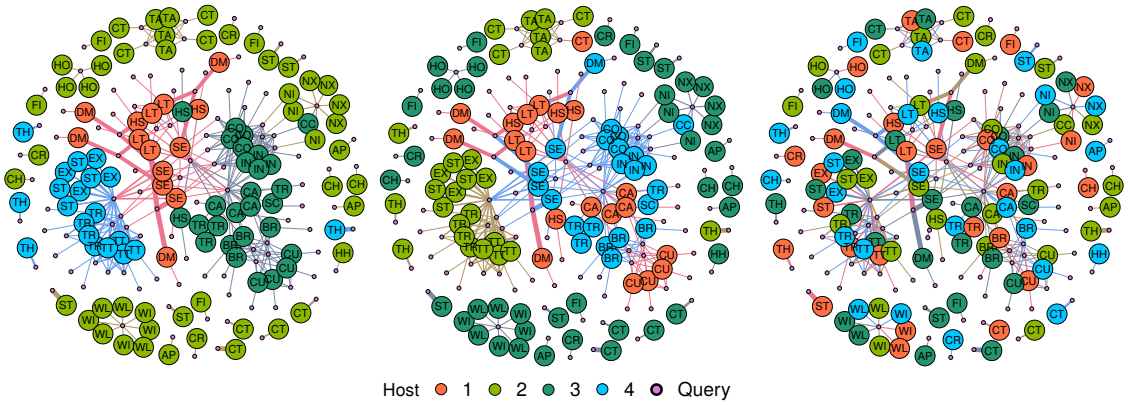


Figure 6.12: Comparison of SWORD, Schism, and Clay: Normalized throughput (current/max) with 0.95 confidence interval after reassignment for TPC-E on 4 hosts



(a) SWORD's workload graph (b) Schism's workload graph (c) Clay's workload graph

Figure 6.13: Comparison of SWORD, Schism, and Clay: Workload graphs after assignment optimization for TPC-E benchmark running on 4 Hosts. The node labels symbolize the table of the shard.

SWORD results in a much higher throughput than using Schism or Clay. Schism results in a speed-up in throughput $S = 1.74$, Clay results in $S = 1.00$ and SWORD in $S = 3.09$.

Figure 6.13 depicts the shard assignment after the optimization. Clay did not move anything because it is designed for workloads with high skews and the blocks are balanced under its balance definition, so it does not reduce any distributed queries. Because Schism uses a graph model instead of a hypergraph model it cuts the wrong edges, resulting in a worse performance than SWORD.

As shown in Figure A.4, the response times for the transaction `MarketWatch` are significantly higher than before the optimization if Schism is used. In contrast, using SWORD the response time is nearly the same after the optimization took place (see Figure 6.9). This is the case because initially most of its queries are non distributed as all shards involved in a query are placed on the same host by the round robin assignment. The involved shards are the `LastTrade (LT)`, `Security (SE)`, and `DailyMarket (DM)` which are placed on different hosts in Schism's optimized assignment, leading to significantly worse overall performance.

6.5. Evaluation of KaDaRea

In this section we evaluate KaDaRea with its novel sliding window approach and compare its results to SWORD. As our approach without sliding window creates the same workload model as SWORD, this sections also compares KaDaRea with our other approach.

We use the same experimental setup and parameters as in the previous section, besides that we configure the imbalance parameter ε to be 0.1 if KaDaRea is used and the TPC-E benchmark is executed. Furthermore, we configure the sliding window to have a size of 50 seconds and a step of 20 seconds.

We reduce the imbalance parameter for KaDaRea because we observed that the imbalance of the assignment increases in some cases, e.g. the imbalance for TPC-C stayed at 0.25 but for TPC-E it increased to 0.52 if an imbalance parameter of $\varepsilon = 0.25$ is used. The imbalance can increase because if a shard is assigned to multiple blocks during KaDaRea, we assign them to the block that has the highest rating. This increases the imbalance by leading to a higher weight of the block it is finally assigned to and a lower weight on the blocks it is not assigned to. To allow a fair comparison between the approaches we configure the imbalance value to be $\varepsilon = 0.1$ instead of $\varepsilon = 0.25$ if KaDaRea is used and the TPC-E benchmark is executed. Thus, the resulting imbalance is between 0.2 and 0.25 in all cases. Future work could look into it and improve our approach to create a more balanced assignment while maintaining the solution quality regarding throughput.

First, we show that KaDaRea performs as good as SWORD for workloads that neither contain peaks nor changing workload patterns. Then we show that KaDaRea performs better in those cases and thus performs also better as our other approach without using a sliding window.

6.5.1. Non Changing Workload

Table 6.4 shows that there is no difference in terms of speed-up in throughput S and share of distributed queries D between SWORD and KaDaRea if the intensity or mixture of the workload does not change. This means that KaDaRea can be generally used even in cases where changing workload patterns and workload peaks are not present.

Table 6.4: Evaluation of KaDaRea with non changing workload regarding speed-up in throughput S and weighted share of distributed queries D on 4 hosts

	TPC-C		TPC-E	
Method	S	D	S	D
KaDaRea	1.75	0.04	3.11	0.10
SWORD	1.78	0.03	3.09	0.11

6.5.2. Peak Workload

If there are peaks in the workload where the load is much higher compared to other parts of the workload we assume that KaDaRea performs significantly better during the peak times. To evaluate this behavior we modified the TPC-C and TPC-E benchmarks to simulate peak times.

We modified the TPC-C benchmark so that a single client runs 4 of the 5 transactions for most of time which simulates the low load that is present most of the day. Then a peak is simulated where the system is fully utilized and clients trigger only the other transaction for 30 seconds.

In a similar way we modified the TPC-E benchmark: A single client runs 7 of 10 transaction types most of the time and during the 30 seconds peak time the system is fully utilized by executing the other 3 transactions.

Figure 6.14 depicts the normalized throughput if the assignment is optimized using SWORD and KaDaRea. Due the low load KaDaRea has less throughput than SWORD which does not optimize for peaks: 33% for TPC-C and 36% less for TPC-E. But, during the peak load KaDaRea processes much more transactions than SWORD: 27% for TPC-C and 42% for TPC-E.

The difference in the optimized shard assignment for TPC-C is shown in Figure 6.15, during the peak the shards `Warehouse (W)` and `Customer (C)` are queried together, thus they are placed on the same host if the assignment is optimized using KaDaRea but not if SWORD is used.

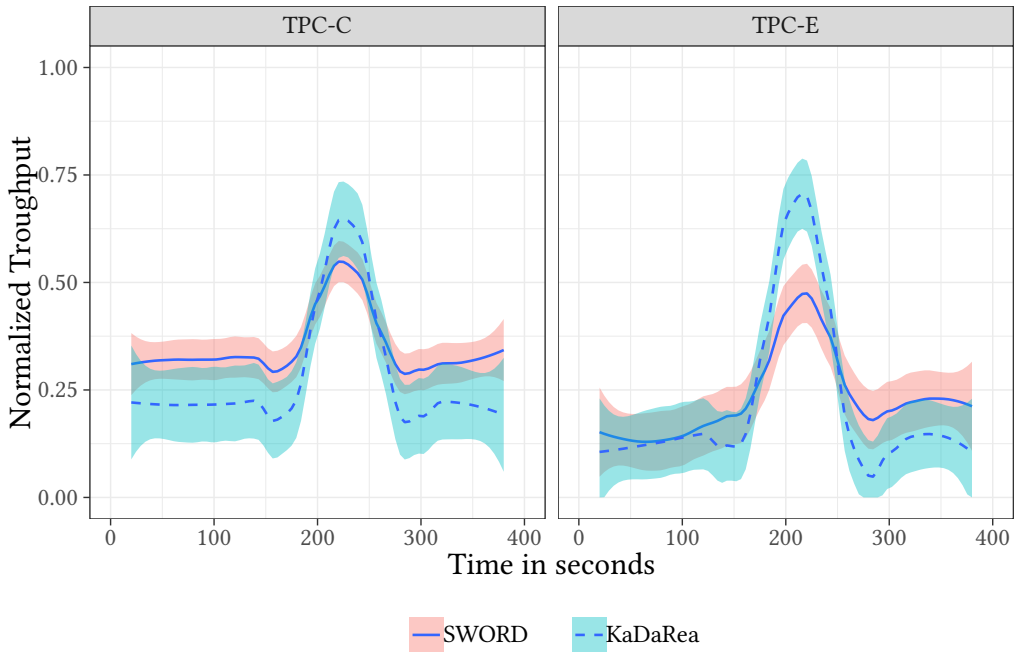
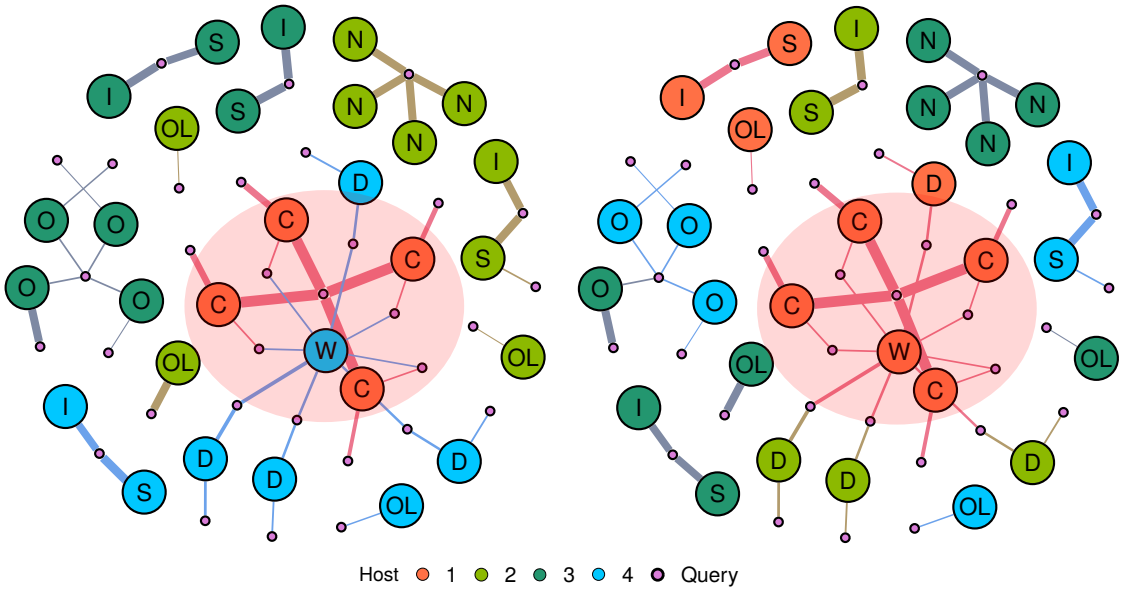


Figure 6.14: Optimizing assignment for peak workload on 4 hosts using modified TPC-C/E benchmarks: Normalized throughput (current/max) with 0.95 confidence interval for SWORD and KaDaRea



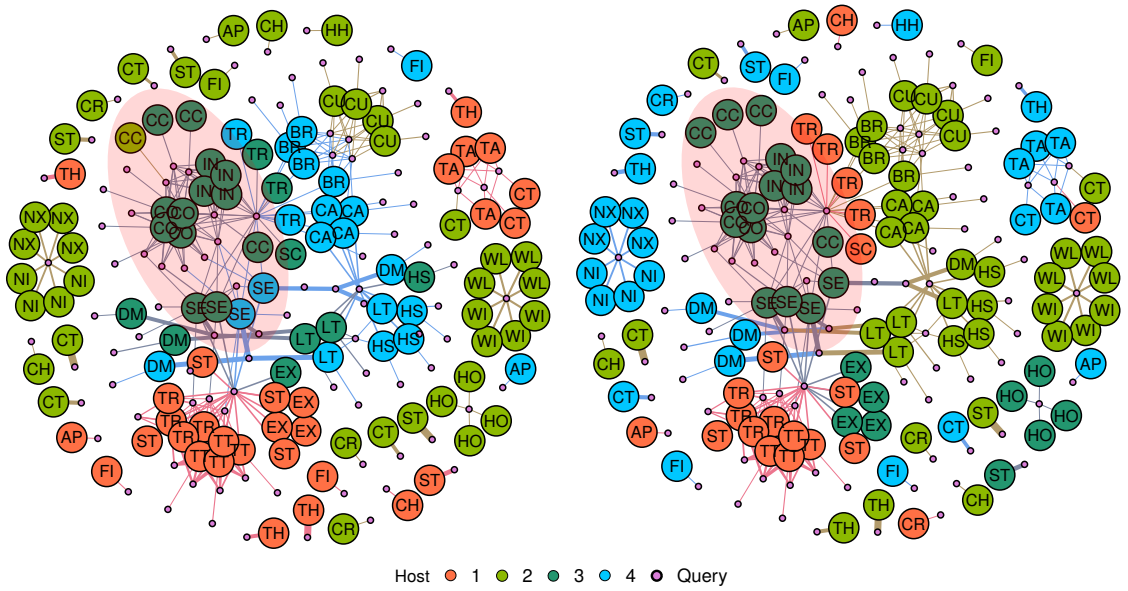
(a) SWORD's workload graph

(b) KaDaRea's workload graph

Figure 6.15: Comparison of workload graphs for peak load using SWORD and KaDaRea for TPC-C. The node labels symbolize the table of the shard.

Figure 6.16 depicts the optimized assignment for TPC-E. During the peak the highlighted shards are queried a lot, which are placed on one host by KaDaRea but on different hosts if SWORD is used.

During low load times the system can handle the load even without advanced reassignment techniques so the smaller throughput speed-up for this time is not a big disadvantage. But more importantly during high load the increased throughput results that the 27% or 42% more customers can use the system without the system getting unresponsive.



(a) SWORD's workload graph

(b) KaDaRea's workload graph

Figure 6.16: Comparison of workload graphs for peak load using SWORD and KaDaRea for TPC-E. The node labels symbolize the table of the shard.

6.5.3. Workload Changing in Patterns

To simulate a multi-tenant workload where teams from different regions are working on the same database, we created a new benchmark by first running the TPC-C benchmark for 300 seconds and afterwards the TPC-E benchmark for 150 seconds.

If the resulting workload hypergraph is partitioned using SWORD, it will place most shards of TPC-C on one host and most of TPC-E on another, in order to eliminate nearly all distributed queries. This can be seen in Figure 6.18 where the highlighted regions are clusters of TPC-C and TPC-E shards. Using SWORD one cluster is placed on one host, leading to no distributed queries. However, this partitioning result is highly imbalanced for all points in time, thus the performance is much worse than partitioning it using KaDaRea, which is depicted in Figure 6.17. As one can see, the throughput of the assignment created by KaDaRea is much higher, because it distributes the shards of both benchmarks on several hosts, resulting on a good utilization of the hosts at any time. During the execution of TPC-C the throughput is a factor of 1.06 higher as SWORD which does not use a using sliding window, while during the execution of TPC-E it is a factor of 1.92 higher.

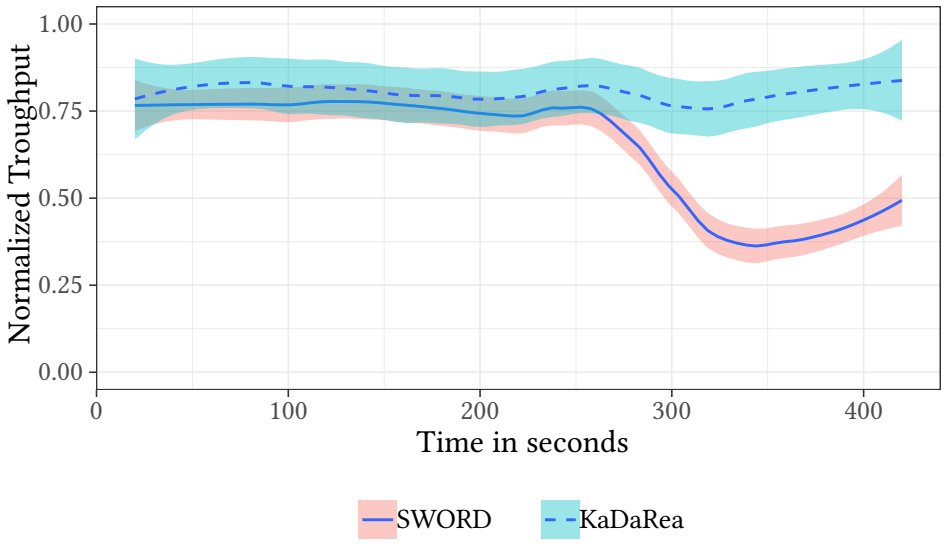
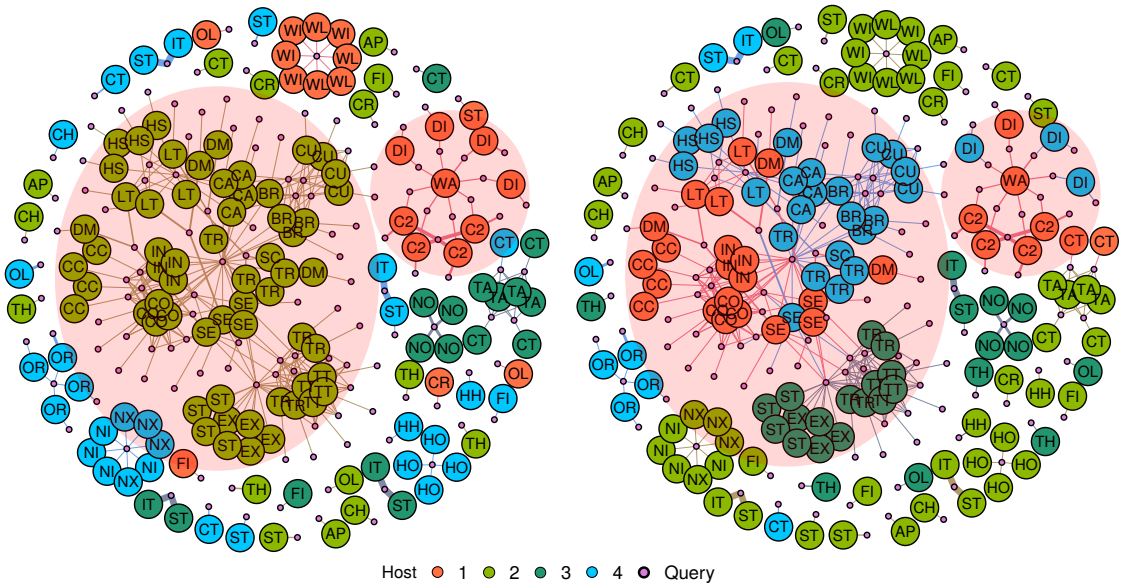


Figure 6.17: Optimizing assignment for workload changing in patterns on 4 hosts: Normalized throughput (current/max) with 0.95 confidence interval for SWORD and KaDaRea.



(a) SWORD's workload graph

(b) KaDaRea's workload graph

Figure 6.18: Comparison of workload graphs for workload changing in patterns for SWORD and KaDaRea. The node labels symbolize the table of the shard.

7. Conclusion

In this thesis we successfully integrated a workload-aware reassignment technique based on hypergraph partitioning for OLTP workloads into the commercial enterprise database system SAP Vora. This technique aims to increase the throughput of the system by reducing the number of distributed queries while keeping the load across the hosts balanced. To realize this, it places frequently co-accessed shards on the same host which eliminates communication overhead and multiple processing of the queries on each involved host.

It is the first time that such an approach is integrated into a commercial enterprise system and it shows a huge performance improvement regarding throughput and response time for the TPC-C and TPC-E benchmarks. For example, the throughput increased 1.78 times on 4 hosts and 1.94 times on 8 hosts for the TPC-C benchmark. For the more complex TPC-E benchmark the speed-up is even larger: 3.09 times on 4 hosts and 5.11 times on 8 hosts. Another insight is that to make Vora scalable, one has to optimize the shard assignment otherwise there is little to no effect of adding hosts to the cluster.

Furthermore, we proposed our novel sliding window based approach for solving the assignment problem, called KaDaRea. KaDaRea considers the time of execution of queries by splitting the workload into time slices and partition each of it. This enables KaDaRea to optimize the assignment for peaks in the workload and to create a better assignment in presence of changes in the workload patterns. The evaluation shows that KaDaRea is generally usable, as it leads to same improvement than an approach that does not use a sliding window for steady workloads, but outperforms other state-of-the-art techniques if peaks or workload changes are present. In these cases, it leads to 46% more throughput during peak times and up to 92% more throughput if the workload patterns change compared to approaches that do not use a sliding window.

Additionally, we experimented with various hypergraph partitioning parameters and evaluated the effects of using different object metrics, weight policies, and imbalance values because previous papers lack a detailed examination of them. These experiments show that using the connectivity metric, the frequency weight policy and an imbalance parameter depending on the number of hosts lead to the best performing assignments. Also, we showed that our approach is able to create good working workload models in the presence of sampling, so we do not need to monitor all queries to build a good performing workload model which could be expensive in some systems.

Finally, we can conclude that optimizing the shard assignment in SAP Vora based on hypergraph partitioning results in a huge performance improvement by increasing the throughput up to 5 times and enables the system to be scalable. Furthermore, KaDaRea outperforms other state-of-the-art techniques for changing workload patterns and workload peaks.

8. Future Work

Future work could focus on the integration of replication into our approach to reduce the number of distributed queries even further. This could be solved creating a number of replicas for each shard depending on how frequent the shard is part of distributed queries and how many hosts they span. These replicated shards are then assigned to hyperedges in the hypergraph model by splitting the heaviest hyperedge that contains the original shard [26]. After partitioning the hypergraph, the original shard is replicated to all hosts which its replicas are assigned to.

Also, instead of triggering the reassignment via a SQL command it could be triggered automatically by Vora. To implement this, one could create a monitor that monitors the number of distributed queries and triggers the reassignment if they cross a certain threshold, like SWORD did [26].

Currently, computing load in terms of frequency of queries is the only constraint when partitioning the hypergraph. Thus, one could add additional constraints like disk space or memory when optimizing the assignment. This leads to the *multi constrained hypergraph partitioning problem* which is already solved in hypergraph partitioning [2].

Additionally, future work could incorporate the migration costs into the hypergraph model to decide if its worth to move a shard. To realize that one could follow the approach of Catalyurek et al. [4] which adds a fixed vertex per host and connects this vertex with all shards that are placed on this host. The weight of these edges represent the data size of a shard and thus the cost of migrating it.

In the evaluation we observed that KaDaRea creates well performing assignments, but it can lead to solutions with a higher imbalance than the configured imbalance parameter ε. To improve our approach KaDaRea could be modified to satisfy this balance constraint. However, the modification should be implemented in a way that does not affect the throughput negatively. For example, implementing a simple greedy algorithm that orders the shards by their rating and assigns shards to hosts as long as the balance constraint is fulfilled would not work out, as groups of shards which are frequently accessed together could then be separated. This would lead to a worse throughput and therefore affect the assignment quality negatively. Additionally, it is questionable if the current imbalance definition is the correct metric when using KaDaRea as the workload changes over time. For example, in Section 6.5.3 we showed that the partitioning result over the complete hypergraph is balanced even if the assignment is not balanced in any point in time. Therefore, we propose an *average imbalance over time* which computes the average imbalance for each time slice.

Finally, future work could also look into the support of OLAP workloads. To support OLAP workloads, one should create a join graph similar to the one presented in Section 3.7 and use it to find a better partitioning of tables. The tables that are most frequent joint are partitioned by their join predicate and co-located across the hosts. This allows parallelized

local processing of queries without communication overhead. Furthermore, one could also create an automated workload classification which decides if the current workload is rather an OLTP or OLAP workload based on the execution time of queries and load of the system because OLAP queries are rather long running and less frequent compared to OLTP queries. This information can then be used to choose the best fitting reassignment strategy and optimize the assignment accordingly.

Bibliography

[1] Yaroslav Akhremtsev et al. “Engineering a direct k-way hypergraph partitioning algorithm”. In: *2017 Proceedings of the Ninteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2017, pp. 28–42.

[2] Cevdet Aykanat, B. Barla Cambazoglu, and Bora Uçar. “Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices”. In: *Journal of Parallel and Distributed Computing* 68.5 (2008), pp. 609–625.

[3] Martin Boissier and Kurzynski Daniel. “Workload-driven horizontal partitioning and pruning for large HTAP systems”. In: *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2018, pp. 116–121.

[4] Umit V. Catalyurek et al. “Hypergraph-based dynamic load balancing for adaptive scientific computations”. In: *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE. 2007, pp. 1–11.

[5] Surajit Chaudhuri and Umeshwar Dayal. “An overview of data warehousing and OLAP technology”. In: *ACM Sigmod record* 26.1 (1997), pp. 65–74.

[6] Shimin Chen et al. “TPC-E vs. TPC-C: characterizing the new TPC-E benchmark via an I/O comparison study”. In: *ACM SIGMOD Record* 39.3 (2011), pp. 5–10.

[7] Yong-Qing Cheng et al. “Maximum-weight bipartite matching technique and its application in image feature matching”. In: *Visual Communications and Image Processing’96*. Vol. 2727. International Society for Optics and Photonics. 1996, pp. 453–463.

[8] Carlo Curino et al. “Schism: a workload-driven approach to database replication and partitioning”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 48–57.

[9] Karen D. Devine et al. “Parallel hypergraph partitioning for scientific computing”. In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE. 2006, 10–pp.

[10] Said Elnaffar and Pat Martin. “Characterizing computer systems’ workloads”. In: *Submitted to ACM Computing Surveys Journal* (2002).

[11] Anil K. Goel et al. “Towards scalable real-time analytics: an architecture for scale-out of OLxP workloads”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1716–1727.

[12] Tobias Heuer. “High quality hypergraph partitioning via max-flow-min-cut computations”. MA thesis. KIT, 2018.

[13] Tobias Heuer, Peter Sanders, and Sebastian Schlag. “Network flow-based refinement for multilevel hypergraph partitioning”. In: *17th International Symposium on Experimental Algorithms (SEA 2018)*. 2018, 1:1–1:19.

[14] Tobias Heuer and Sebastian Schlag. “Improving coarsening schemes for hypergraph partitioning by exploiting community structure”. In: *16th International Symposium on Experimental Algorithms (SEA 2017)*. 2017, 21:1–21:19.

[15] T.C. Hu and K. Moerder. “Multiterminal flows in a hypergraph”. In: *VLSI circuit layout: theory and design* (1985), pp. 87–93.

[16] Edmund Ihler, Dorothea Wagner, and Frank Wagner. “Modeling hypergraphs by graphs with the same mincut properties”. In: *Information Processing Letters* 45.4 (1993), pp. 171–175.

[17] George Karypis et al. “Multilevel hypergraph partitioning: applications in VLSI domain”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 7.1 (1999), pp. 69–79.

[18] Naoki Katoh, Akiyoshi Shioura, and Toshihide Ibaraki. “Resource allocation problems”. In: *Handbook of combinatorial optimization* (2013), pp. 2897–2988.

[19] Harold W. Kuhn. “The Hungarian method for the assignment problem”. In: *Naval research logistics quarterly* 2.1-2 (1955), pp. 83–97.

[20] Thomas Lengauer. *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

[21] Scott T. Leutenegger and Daniel Dias. *A modeling study of the TPC-C benchmark*. Vol. 22. 2. ACM, 1993.

[22] Yoon-Min Nam, Min-Soo Kim, and Donghyoung Han. “A Graph-based Database Partitioning Method for Parallel OLAP Query Processing”. In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. IEEE. 2018, pp. 1025–1036.

[23] Oracle. *Database VLDB and partitioning guide*. URL: `https://docs.oracle.com/database/121/VLDBG/toc.htm` (visited on 03/26/2019).

[24] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[25] Andy Pavlo. *H-Store documentation*. URL: `http://hstore.cs.brown.edu/documentation/` (visited on 03/20/2019).

[26] Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. “SWORD: scalable workload-aware data placement for transactional workloads”. In: *Proceedings of the 16th International Conference on Extending Database Technology*. ACM. 2013, pp. 430–441.

[27] Sebastian Schlag et al. “k-way hypergraph partitioning via n-level recursive bisection”. In: *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2016, pp. 53–67.

[28] SAP SE. *SAP Vora developer guide*. URL: https://help.sap.com/doc/d873bf7ac8e14a7eb7a1ab686d3028a7/2.4.latest/en-US/loiod873bf7ac8e14a7eb7a1ab686d3028a7.pdf (visited on 03/19/2019).

[29] Marco Serafini et al. "Clay: fine-grained adaptive partitioning for general database schemas". In: *Proceedings of the VLDB Endowment* 10.4 (2016), pp. 445–456.

[30] TPC. *TPC-C database schema*. URL: http://www.tpc.org/information/sessions/sigmod/img009.jpg (visited on 03/21/2019).

[31] Ata Turk et al. "Temporal workload-aware replicated partitioning for social networks". In: *IEEE Transactions on knowledge and Data Engineering* 26.11 (2014), pp. 2832–2845.

[32] Wenyin Yang et al. "HEPart: A balanced hypergraph partitioning algorithm for big data applications". In: *Future Generation Computer Systems* 83 (2018), pp. 250–268.

[33] Xiaoyan Yang, Cecilia M. Procopiuc, and Divesh Srivastava. "Summarizing relational databases". In: *PVLDB* 2 (2009), pp. 634–645.

[34] Boyang Yu and Jianping Pan. "Location-aware associated data placement for geo-distributed data-intensive applications". In: *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE. 2015, pp. 603–611.

A. Appendix

A.1. Database Schema of TPC-E

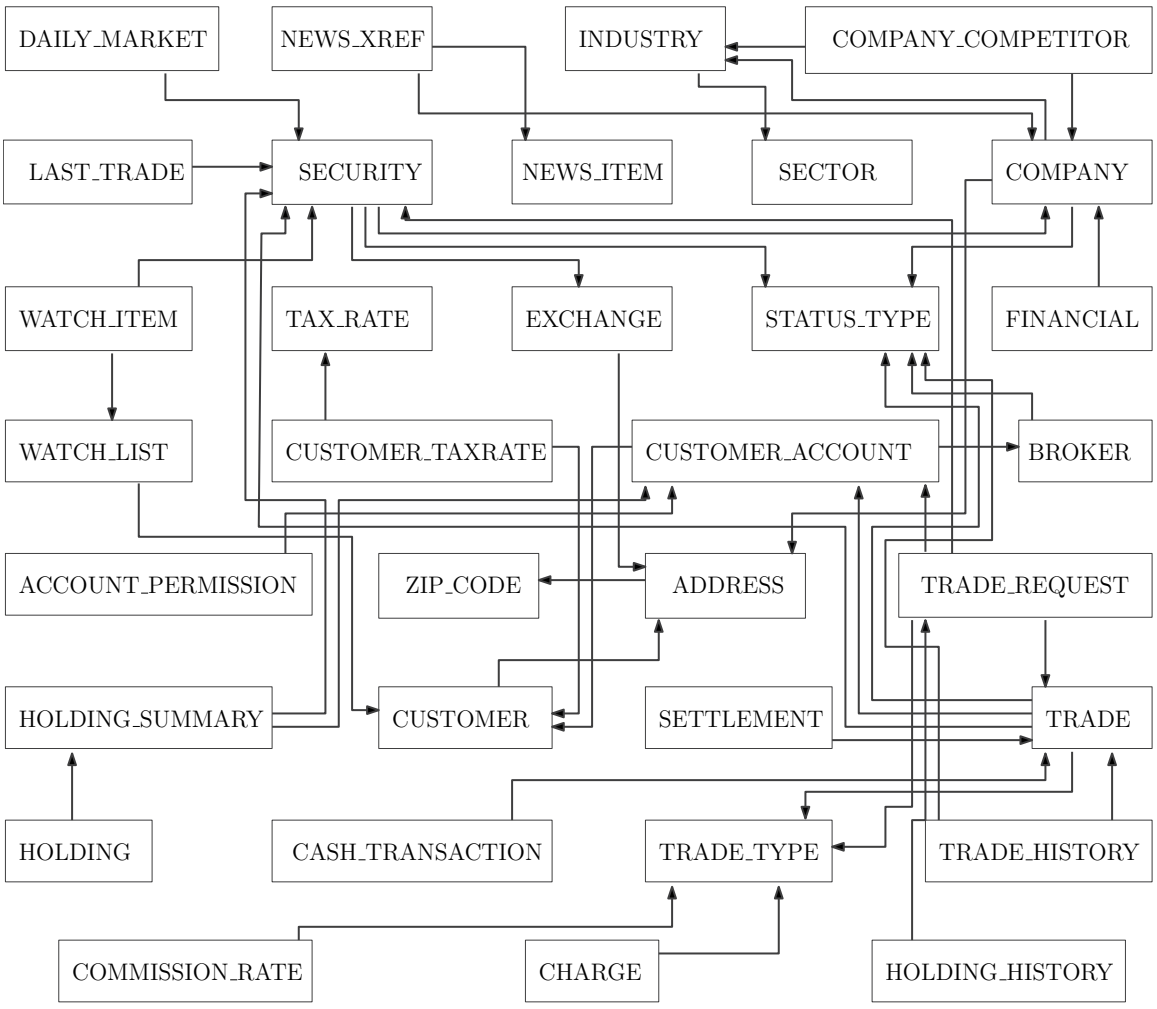


Figure A.1: TPC-E database schema (adapted from [33])

A.2. Distributed Query Plots

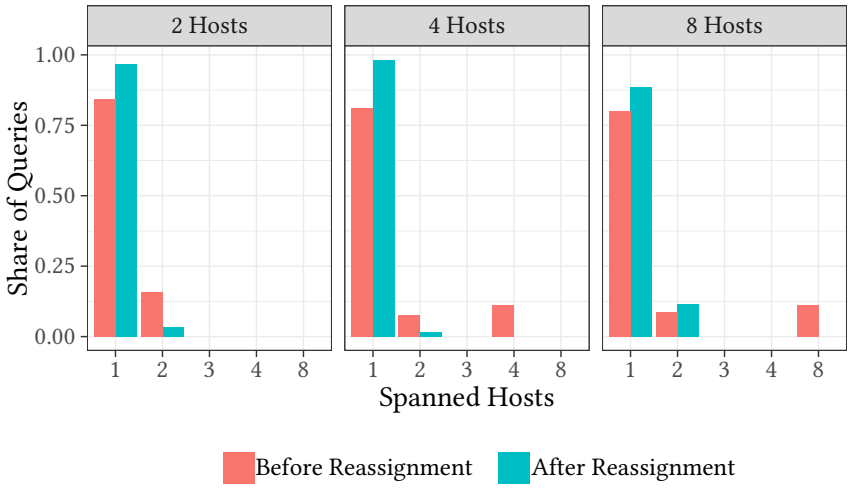


Figure A.2: Share of distributed queries before and after reassignment for TPC-C

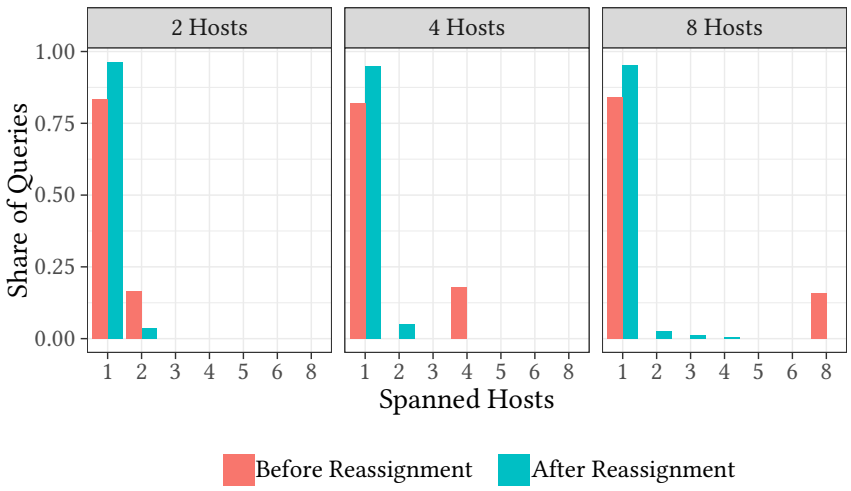


Figure A.3: Share of distributed queries before and after reassignment for TPC-E

A.3. Additional Response Time Plots

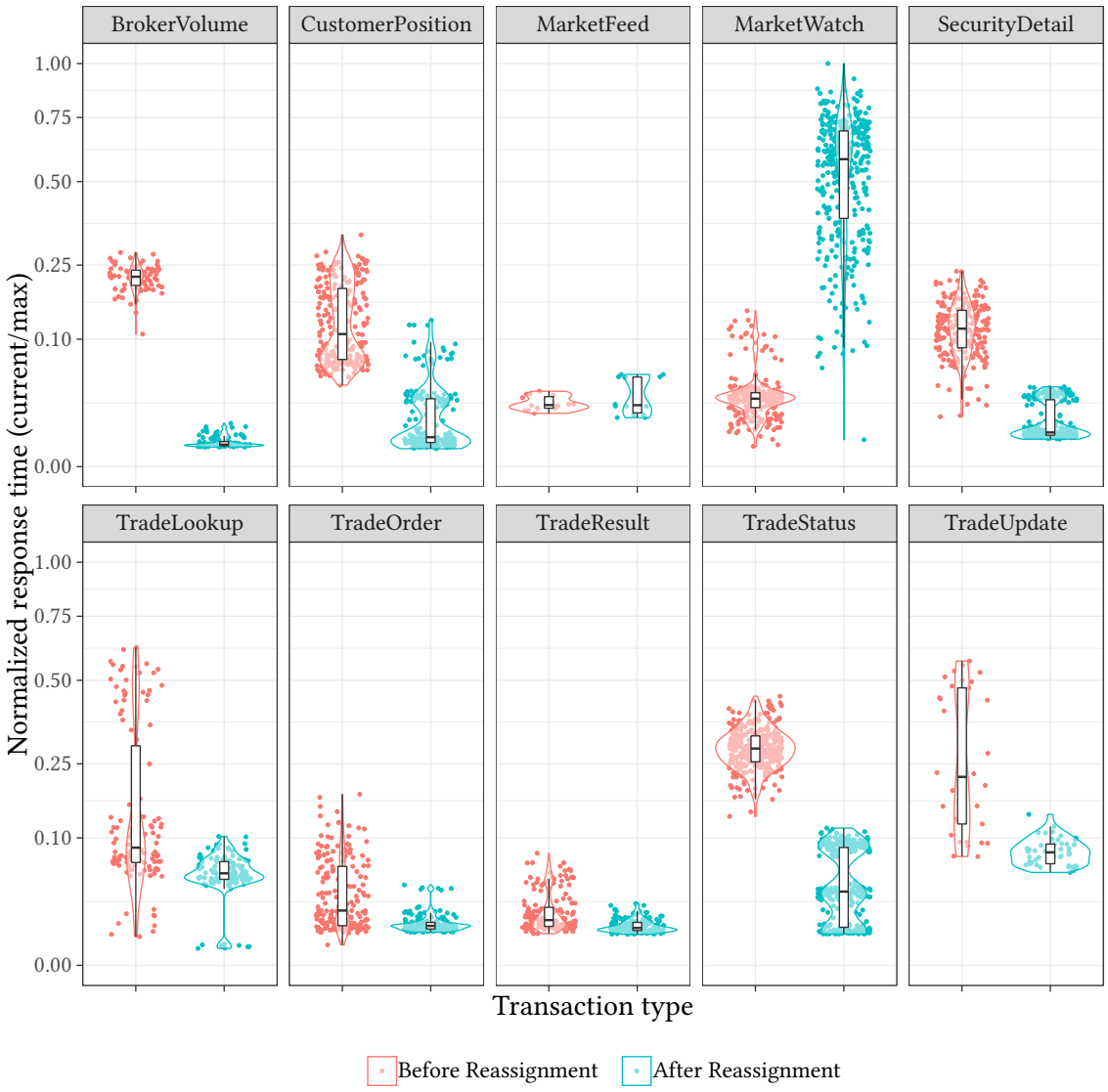


Figure A.4: Normalized response times (current/max) for each transaction type of TPC-E using Schism on 4 hosts

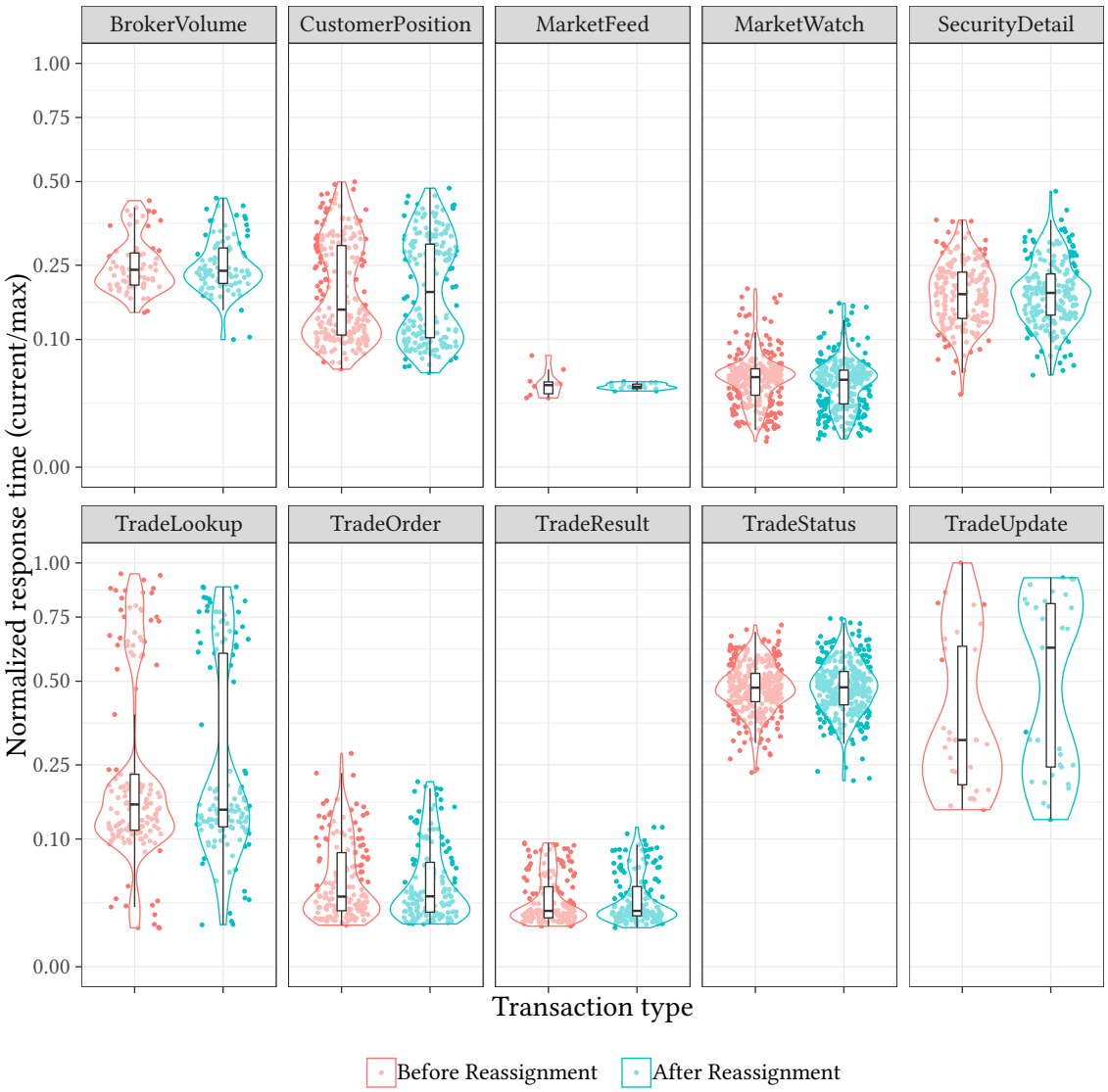


Figure A.5: Normalized response times (current/max) for each transaction type of TPC-E using Clay on 4 hosts

A.4. Increased Shard Number for Database Partitioning

We experimented with higher numbers of shards for database partitioning. This should give the hypergraph partitioner the possibility to move shards more fine granularly resulting in a reduction of distributed queries and an improvement of the performance.

To evaluate this effect, we executed the TPC-E benchmark on 4 hosts, with an ε of 0.25 and partitioned each table into 8 shards instead of 4. Compared to 4 shards per table, the increased shard number leads to a further reduction of the weighted distributed queries D of 40%. However, the overhead created by the additional shards leads to a decrease of the performance, as Figure A.6 depicts.

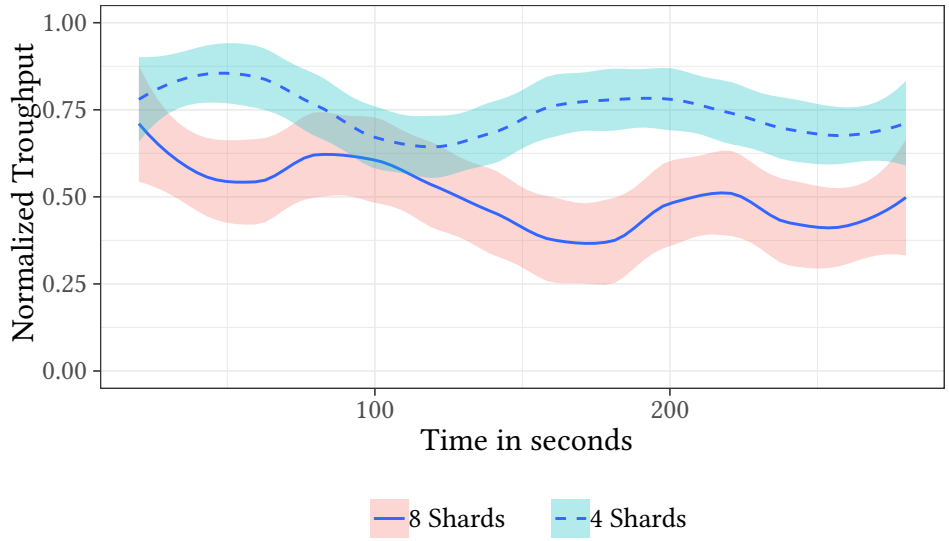


Figure A.6: Normalized throughput (current/max) with 0.95 confidence interval for tables partitioned into 4 and 8 shards on 4 hosts

A.5. Reassignment Time

Figure A.7 depicts the time it at takes to reassign the shards. As one can see, KaHyPar finishes nearly immediately and it takes the most time to move the shards in Vora. Also, the total time is much higher for the TPC-E benchmark as it consists of more tables and thus there are more shards to be moved. Furthermore, the total time increases with increasing hosts because more hosts result in more shards.

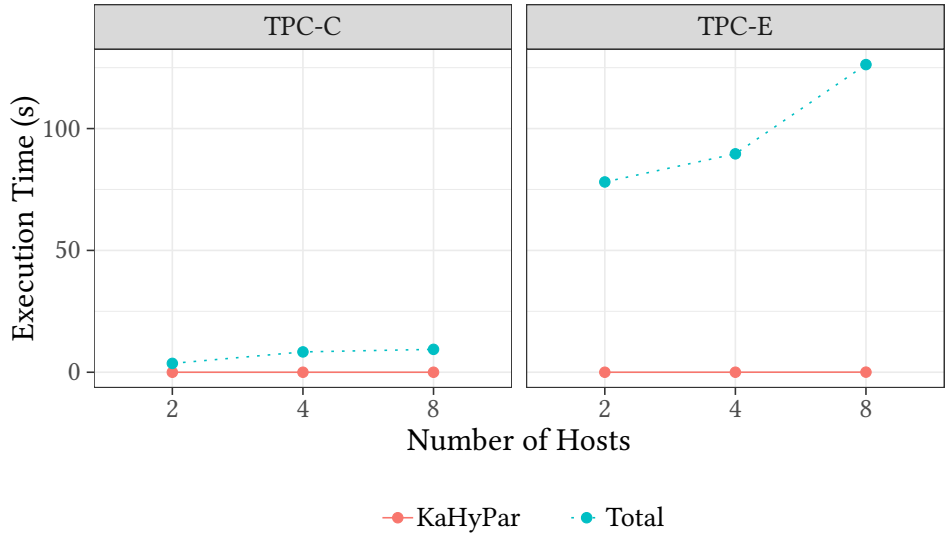


Figure A.7: Reassignment times for 2, 4, and 8 hosts