Journal of the Brazilian Computer Society, 2026, 32:1, doi: 10.5753/jbcs.2026.5684

An Autonomous Hybrid Data Partitioning Approach for NewSQL Databases

Geomar A. Schreiner [**Federal University of Fronteira Sul** | ***gschreiner@uffs.edu.br***]
Rafael de Santiago [**Federal University of Santa Catarina** | ***r.santiago@ufsc.br***]
Denio Duarte [**Federal University of Fronteira Sul** | ***duarte@uffs.edu.br***]
Ronaldo dos Santos Mello [**Federal University of Santa Catarina** | ***r.mello@ufsc.br***]

Universidade Federal da Fronteira Sul, SC-484, Km 02 - Fronteira Sul, Chapecó - SC, 89815-899, Brazil

Received: 13 March 2025 • **Accepted:** 26 August 2025 • **Published:** 02 February 2025

Abstract. Like online games and the financial market, several applications require specific data management features such as large data volume support, data streaming, and the processing of thousands of OLTP transactions per second. In general, traditional relational databases are not suitable for these requirements. NewSQL is a new generation of databases that combines high scalability and availability with ACID support, being a promising solution for these kinds of applications. Although data partitioning is an essential feature for tuning relational databases, it is still an open issue for NewSQL databases. This paper proposes an automated approach for hybrid data partitioning that minimizes the number of distributed transactions and keeps the system well-balanced. In order to demonstrate its efficacy, we compare our solution with an optimal partitioning solution generated by a solver and a state-of-art baseline. The experiments show that the quality of the partitioning scheme is similar to the optional solution and overcomes the state-of-art approach in number of distributed transactions.

Keywords: NewSQL, Heat Graph, Data Partition, Hybrid Data Partitioning

1 Introduction

Historically, application systems rely on Online Transactional Processing (OLTP) to perform small operations over the network, such as buying an item in an online store [Stonebraker, 2012]. The number of users that use online stores has increased, along with it, the number of OLTP transactions. With the emergence of the *Web*, OLTP requests have changed to deal with transactions on online games, social networks, or even large financial companies. These applications are characterized by having a large number of user interactions, generating a vast amount of data, and multiple OLTP transactions per second.

For decades, traditional Relational Databases (RDBs) have been used as an efficient way to store and handle operational data. However, they are not suitable to deal with massive data volumes with high availability, known as Big Data, and ACID support guarantees [Stonebraker, 2012]. New DB systems to handle Big Data needs are emerging, like NoSQL DBs, to handle the problems to meet the Big Data needs. These new architectures had mainly appeared into cloud environments, which are able to store and manage massive data volumes with high scalability.

Nevertheless, NoSQL DBs solve part of the problems related to Big Data management. They maximize availability rather than ACID support. Due to this, several companies continue to use RDBs for their data-centric applications because they are less expensive to deal with traditional ACID assurance overhead than the lack of these properties. In order to cope with this limitation, the *NewSQL* movement has arisen.

NewSQL DB main advantage is to combine the best of both worlds: the scalability and availability of NoSQL DBs with the ACID support of traditional RDB [Stonebraker, 2012]. Usually, NewSQL DBs are distributed in-memory DBs, and each node/site of the DB holds a partition of the stored data [Pavlo and Aslett, 2016; Taft *et al*., 2014; Elmore *et al*., 2015; Kallman *et al*., 2008]. Because of this, they can handle thousands of OLTP transactions per second. Nevertheless, they suffer when multi-site (distributed) transactions are required, since these transactions are usually serialized and further executed, when it is possible, as single-site. This strategy enables them to use lighter and lock-free concurrency control protocols, but when a transaction needs to access data in multiple sites, the system performance is degraded.

Partitioning is a well-known technique that can be used to mitigate performance degradation generated by multi-site transactions. Each partition stores co-related data, and so transactions can run locally. Relational data are usually partitioned in two ways: *horizontally* (by rows) or *vertically* (by columns) [Al-Kateb *et al*., 2016]. Horizontal partitioning divides a table into groups of tuples (*i.e*, selection operation). Vertical partitioning divides a table into groups of disjoint columns (*i.e*, projection operation), and each group becomes a partition. Vertical partitioning is suitable to OLAP applications since it keeps grouped only the columns (attributes) that are relevant for certain analytical queries, which reduces the cost mainly of join operations. Horizontal partitioning is mainly considered by OLTP solutions, which may update any table column, and all the commercial NewSQL DB systems apply this strategy instead[Al-Kateb *et al*., 2016; M. Tamer Özsu, 2011]. A few academic NewSQL DB systems explore vertical partitioning as an option to reduce the number of distributed transactions in OLTP-based sys-

tems [Amossen, 2010; Schreiner *et al*., 2019], as well as hybrid partitioning for OLAP transactions [Arulraj *et al*., 2016]. However, data partitioning remains an open issue for NewSQL DB systems.

H-Store [Kallman *et al*., 2008] and *S-Store* [Cetintemel *et al*., 2014] are NewSQL DB systems that consider a pre-workload plan to define an optimized partitioning. However, the data volume and workload of a DB are not static, and they do not consider the evaluation and modification of their partitioning strategies. Other proposals, such as *Clay* [Serafini *et al*., 2016], *Accordion* [Serafini *et al*., 2014] and *E-Store* [Taft *et al*., 2014] provide repartitioning tools. All of them have an autonomous system that considers OLTP loads. However, none of them consider a hybrid data partitioning, that is, choosing vertical or horizontal partitioning based on the workloads.

This paper proposes a new automated partitioning system that evolves the partitioning scheme of a NewSQL DB. We consider hybrid data partitioning for minimizing distributed transactions on OLTP systems. Our goal is to propose a reactive system that is able to generate new data partitioning schemes based on the system workload. Additionally, it reorganizes data with no downtime in order to maintain good performance for the NewSQL DB system.

The main contributions of our work are: (*i*) an adaptive approach that allows the DB system to reorganize data based on the current workload; (*ii*) an optimal solution for the data partitioning problem that uses a solver to generate the optimal data partitioning based on a set of objective functions; (*iii*) an approach for data partitioning based in heuristics that considers hybrid data partitioning for OLTP transactions to minimize distributed transactions; (*iv*) a set of experiments that compare our solution with an optimal partitioning scheme generated by a solver using the same goal equations that we considered in our algorithm; and, (*v*) a set of experiments comparing the number of distributed transactions of the original partition scheme, a partition scheme generated by our approach, and a partition scheme generated by a state-of-art approach.

The rest of this paper is organized as follows. Section 2 discusses related work. The Section 3 provides the theoretical foundation: an overview of data partitioning and NewSQL DBs architectures. Section 4 presents the formalization of the heat graph considered by our work. In Section 5 we formalize the graph partitioning problem and present two solutions: (i) the solver solution that represents the optimal one Section 6 demonstrates the partition quality and efficiency of our proposed solution through a set of experiments. At last, Section 7 presents the conclusions and some future works.

2 Related Work

There are some approaches in the literature that propose partitioning solutions for NewSQL DBs, and each one focuses on different perspectives of the problem. H-Store [Kallman *et al*., 2008], the first proposed NewSQL DBMS, focused on the system architecture and not on the data partitioning efficient management. Approaches like *Horticulture* [Pavlo *et al*., 2012], *Schism* [Curino *et al*., 2010], *Accordion* [Serafini *et al*., 2014] and *Clay* [Serafini *et al*., 2016] present data partitioning techniques for reducing the number of distributed transactions. *E-Store* [Taft *et al*., 2014] and *S-Store* [Meehan *et al*., 2015] also propose data partitioning techniques, but with specific purposes: scalability and streaming. The *Hybrid VoltDB* [Schreiner *et al*., 2019] explores the use of hybrid data partitioning to avoid distributed transactions.

Table 1 summarizes the features analyzed for the related work: *(i)* the considered type of data partitioning (*Part. Type*); *(ii)* an automated solution to create/reorganize the partitions (*Automated Part.*); *(iii)* migration of data between partitions without downtime (*Online Part.*); *(iv)* replication support in order to increase performance (*Replication*); and *(v)* the granularity of the data considered for the data partitioning (*Granularity*).

Almost all approaches that implement partitioning approaches for OLTP-based systems consider horizontal partitioning, *i.e.*, they all split the tables into sets of tuples, and each partition maintains a subset of the table. *H-Store* [Kallman *et al*., 2008], *Horticulture* [Pavlo *et al*., 2012], *Accordeon* [Serafini *et al*., 2014], and *S-Store* [Meehan *et al*., 2015] divide their tuples horizontally using the identifier of each tuple. Each partition stores a set of tuples with sequential identifiers. On the other hand, *Schism* [Curino *et al*., 2010], *E-Store* [Taft *et al*., 2014], and *Clay* [Serafini *et al*., 2016] group tuples by affinity, *i.e.*, tuples accessed together most of the time are stored in the same partition. Additionally, the *E-Store* uses vertical partitions to store data used in streamings, but they are replicated from the original data and are just used as read-only data. *Hybrid VoltDB* is the only proposal that considers hybrid data partitioning, which sounds promising to reduce the number of distributed transactions.

H-Store, *S-Store*, and *Hybrid VoltDB* requires the intervention of a DBA or another external tool to create a suitable partitioning for the desired demands. On the other hand, most of the approaches support an automated partitioning solution aiming at decreasing the distributed transactions. *Horticulture* takes into account the operations provided by the DBA, as well as the data schema, to feed its *Large-Neighborhood Search (LNS)* algorithm, which generates the partitioning. *E-Store* partitions its data using some statistics to identify the most accessed tuples. These tuples are allocated in different vertexes to balance the load. *Accordion*, in addition to data statistics generated from the workload, takes into account the maximum capacity of each server and groups the partitions accessed together on the same server or nearby servers. Different from them, *Clay* and *Schism* use a *heat graph* to accomplish this task. This graph is also created based on the tuple accesses. Each vertex represents a tuple accessed by a transaction, maintaining the number of times the tuple was accessed. Edges connecting vertices represent tuples accessed together by the same transaction.

Only four approaches use replicas of data to improve partitioning performance. *Horticulture* and *Schism* create a partitioning plan that takes into account block replication to facilitate transaction execution and increase system availability. *S-Store* does not have a re-partitioning model, but it performs data replication for *streaming* support. Data accessed by a *streaming* operation are replicated into tables created exclusively for this purpose. The *Hybrid VoltDB* does not consider

Table 1. Related work comparison

Approach	Part. Type	Automated Part.	Online Part.	Replication	Granularity
H-Store (2008)	Horizontal	No	No	-	-
Horticulture (2012)	Horizontal	Yes	No	Yes	Blocks
Schism (2012)	Horizontal	Yes	No	Yes	Tuple
E-Store (2014)	Horiz./Vert	Yes	Yes	No	Tuple
Accordeon (2014)	Horizontal	Yes	Yes	No	Blocks
S-Store (2015)	Horizontal	No	No	Yes	-
Clay (2016)	Horizontal	Yes	Yes	No	Tuple
Hybrid VoltDB (2019)	Hybrid	No	Yes	Yes	Tuple/Atributes

an automated system that creates a replication-based strategy. Instead, it allows the DBA to define a partitioning plan that uses replication as an additional feature to reduce the number of distributed transactions.

From the analyzed approaches, most of them work only with two levels of granularity: blocks and tuples. *Horticulture* and *Accordion* use a larger granularity than the others, working directly with data blocks (sets of tuples). These blocks are created using a classic partitioning strategy (*e.g.*, grouping and dividing data using a *hash* for each tuple). *E-Store* and *Clay* use a finer granularity, working directly with tuples. This granularity facilitates the migration of tuples between partitions. Approaches that use a finer granularity can separate tuples that are widely accessed and have no links, or even change partitions by placing tuples that are widely accessed in the same partition. However, generally, the management and statistics required by a finer-grained approach are more complex [M. Tamer Özsu, 2011]. *Hybrid VoltDB* applies an attribute/tuple granularity over the VoltDB structure. It does not map where each tuple is located in the cluster. The statements are executed in a broadcast for all sites, and only sites that have the data perform the operation. This strategy simplifies the statistics required to maintain the fine granularity.

From this related work analysis, we can see the lack of an approach that explores some specific aspects: *(i)* a hybrid data partitioning that decides when it is advantageous to use vertical or horizontal partitioning; *(ii)* a solution that considers replicas to increase availability and, therefore, decreases the number of distributed transactions; and *(iii)* a solution that performs a gradual and periodic evolution of the existing partitions, avoiding periods of high latency until its partitioning is optimized.

The approach proposed in this paper deals with the aspects *(i)* and *(iii)* with a fine grain granularity (tuple/attribute). It is detailed in the next sections. It considers a heat graph as a tool to accomplish data partitioning, as described in the following.

3 Theoretical Foundation

This section gives a brief background about data partitioning and NewSQL DBs. These concepts are necessary to properly understand our solution.

3.1 Data Partitioning

Data partitioning is defined as the division of a table into multiple independent parts (partitions). Database partitioning is usually applied as an optimization for data accessing since dividing the amount of data can increase manageability, availability, and load balancing. On considering a distributed RDB system: *(i)* a table can be stored in a single site, or *(ii)* a strategy can be created for distributing the table fragments among multiple sites.

The first option generates a high number of additional remote data access required to process join operations [M. Tamer Özsu, 2011]. Additionally, the maintenance of ACID properties becomes more complicated. Moreover, replicating all the tables in every site generates high data redundancy, leading to data consistency and data storage misused on the sites. Thus, it is natural to consider the second option. A table decomposed into fragments allows several performance optimizations, like local join execution [M. Tamer Özsu, 2011].

Partitioning techniques can be divided into three categories: *horizontal*, *vertical*, and *hybrid* [Sharma and Kaur, 2015]. *Horizontal* partitioning fragments a relation into other subrelations (a subset) [Sharma and Kaur, 2015]. A horizontal fragment of a relation R consists of all R tuples that satisfy a partitioning function f_i [M. Tamer Özsu, 2011]. This function f_i can be developed as an automated process that analyzes the DB schema and statistics about data accessing inferring groups of rows by affinity or manually using SQL DDL. Based on the f_i result, tuples are routed directly to specific sites. Figure 1 (A) shows a horizontal partitioning defined for a $Movies$ table. It was split by column *id*. Blue lines belong to the partition P_1 ($f_i(id\%2 = 1)$) and yellow lines to the partition P_2 ($f_i(id\%2 = 0)$).

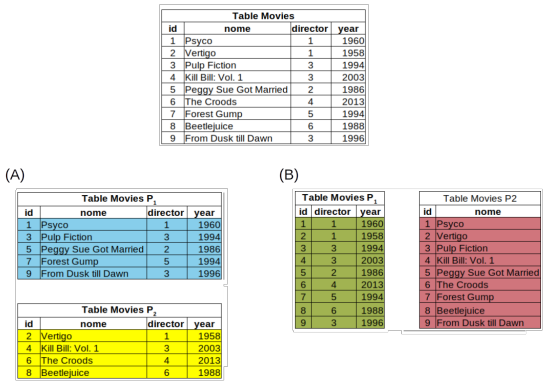

Figure 1. Data partitioning: (A) horizontal partitioning; (B) Vertical partitioning [Schreiner *et al.*, 2019]

One way to fragment data in horizontal partitioning is using a *range* function, and this approach defines categories and stores them on different nodes [Grolinger *et al*., 2013; Sharma and Kaur, 2015]. One node is responsible for, based on a *range* function, assigning tuples to other nodes and maintaining mappings among the sites to locate data. However, the *range* approach might cause a load unbalance between nodes because some categories may contain more tuples than others. So, another approach is to consider a *Hash* function. In *hash* partitioning, the system usually assumes a ring organization of the system nodes, and each node is responsible for a set of keys. In order to find a tuple t_i location, t_i key is submitted to the *hash* function. With this approach, t_i is quickly located in the ring, so there is no need to maintain mappings.

The vertical partitioning, on the other hand, considers fragments of a relation R as projections R_1, R_2, ..., $R_n \subseteq R$ over different sets of attributes [M. Tamer Özsu, 2011; Navathe *et al*., 1984]. The main idea is to group attributes into fragments that are often accessed together by applications [Sharma and Kaur, 2015]. However, this partitioning approach is complex since each attribute that composes one fragment directly affects the system's performance. Figure 1 (B) shows an example of two vertical partitions for the table *Movies*. Notice that the column *id* is presented in both fragments. It represents the primary key and must be replicated in each fragment to allow the tuple reconstruction.

For some applications, horizontal or vertical partitioning of a DB is not sufficient [M. Tamer Özsu, 2011]. So, a *hybrid* partitioning is needed. In this case, horizontal and vertical partitioning are simultaneously applied over the DB. Each fragment maintains a set of rows with a specific set of columns stored in a node, being useful for queries that intend to retrieve such a table subset.

On considering the table Movie, we apply the horizontal partitioning in Figure 1(A) dividing the table into two partitions. Otherwise, in Figure 1(B) we split the table into two vertical partitions: the first one with 3 columns and the second one with two. Therefore, hybrid data partitioning is a set of horizontal fragments that are further partitioned into vertical fragments. In the example, a hybrid data partitioning could create the following partitions: (*i*) one with five rows (Figure 1(A) blue rows) and three columns (*id, director* and *year*); (*ii*) one with five lines (Figure 1(A) blue rows) and two columns (*id* and *name*), being similar to Figure 1(B) but with less tuples.

The use of horizontal partitioning causes a distributed transaction when a query requires all the data of a specific column. As the data is divided into subsets of rows, all nodes must be consulted to retrieve the data. Vertical partitioning solves this problem by storing data from a column in just one node. However, by using only vertical partitioning, the chances of a load unbalance occurs are very high since if a column is often accessed, the system will always access the same site since it stores all the rows that contain that column. Hybrid partitioning, in turn, provides a finer adjustment in data storage, allowing load balancing of the horizontal model and also storing commonly accessed columns together on the same site.

Despite these advantages, the use of hybrid partitioning accentuates the complexity of defining an appropriate partitioning. In addition, the allocation of partitions also has increased complexity, because depending on their position in the cluster, the number of distributed transactions can increase. In this work, we use a heat graph to represent interactions (accesses) between tuples to address this problem. Based on this graph, we propose the usage of hybrid partitioning to decrease the number of distributed transactions allocating often accessed data in the same site to keep the system balanced.

3.2 NewSQL

NewSQL is a new generation of modern RDBs that supply the demand for large OLTP workloads without giving up on the ACID properties [Stonebraker, 2012; Pavlo and Aslett, 2016; Kumar *et al*., 2014]. As stated before, this family of DBs transcends the traditional RDBs by incorporating essential features to *Big Data* management, such as high availability and scalability [Grolinger *et al*., 2013]. In addition to offering SQL as the primary access language, as well as ACID guarantees, a DB needs to address some other characteristics to be considered a NewSQL DB [Pavlo and Aslett, 2016].

Traditional RDBs use full disk persistence (*i.e.*, they are disk-oriented), and the main memory is considered to optimize some tasks. Instead, NewSQL DBs are usually *in-memory* DBs. They consider the main memory (RAM) as their principal storage space, eliminating the hard drive data accesses overhead. The usage of an *in-memory* approach avoids mechanisms that deal with the constant exchange of data between disk and main memory.

The NewSQL DBs are not the first DB technology that employs main memory as main storage space [Valdes *et al*., 1984; DeWitt *et al*., 1984]. However, they differ from the other ones by reallocating a subset of the DB to persistent memory to reduce its memory consumption [Ma *et al*., 2016]. This technique allows the DBMS to support a larger chunk of data than the total available main memory without using virtual memory as drive-oriented architectures. The most common approach to manage the main memory is to create mechanisms to identify not frequently accessed tuples and reallocate them from memory to the disk.

The first NewSQL DBMS was the *H-Store* [Kallman *et al*., 2008]. All the NewSQL systems are inspired by the H-Store architecture (Figure 2). A NewSQL DB is, essentially, a distributed database, with multiple nodes connected as a cluster and a focus on OLTP transactions. Figure 2 shows multiple H-Store nodes. Each node represents a machine running the H-Store system. Unlike traditional systems, in a NewSQL system, each node is configured to use multiple cores, and each core runs as an independent part of the system called *site*. Each site is composed by an *execution engine* and a *data partition*. The *execution engine* is responsible for executing the necessary operation (*e.g.*, queries, stored procedures) over the data partition (portion of the data).

For NewSQL DBs a key concept is data partitioning, since they affect the performance of the system [Pavlo and Aslett, 2016]. In this case, DB tables are usually divided horizontally into multiple partitions [Al-Kateb *et al*., 2016]. A module of the DBMS assigns each tuple to a fragment based on the values of a set of attributes (possibly unitary) using a range or hash function. As we stated before, partitioning is a well-

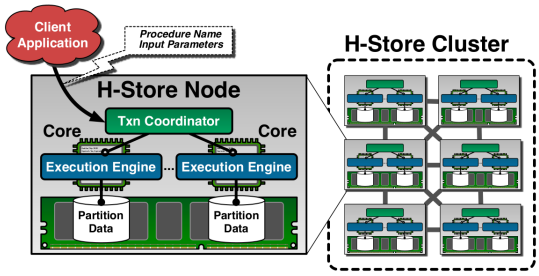

Figure 2. H-Store architecture [Pavlo *et al*., 2011]

known technique that can be used to mitigate the performance degradation caused by distributed transactions. Since the NewSQL DBs holds a shared-nothing distributed architecture, the allocation of data that are accessed together in the same site is very important. If a transaction accesses data stored into only one site, the transaction is executed in a serial way, with no lock. On the other hand, if a transaction touches data allocated in more than one partition (a distributed transaction), the system needs to elect a coordinator that will ensure the ACID properties. The use of a coordinator leads the system to lock situations, reducing the performance of the system.

NewSQL DBMSs ensure high data availability and durability, supporting strongly consistent replication [Pavlo and Aslett, 2016]. The replication methods generally consider a wide-area network (WAN) replication. Any NewSQL DBMS can be configured to provide synchronous WAN data updates, but it can cause significant slowness in normal operations [Schreiner *et al*., 2019]. Thus, they provide asynchronous replication methods.

A critical feature of NewSQL DBs is the concurrency control [Pavlo and Aslett, 2016]. Concurrency control allows multiple users to access the DBs at the same time and execute their transactions independently. There are two methods of concurrency control mainly used by NewSQL DBs: *TO (timestamp ordering)* and *MVCC (multi-version concurrency control)*. A few ones use the TO [Schreiner *et al*., 2019]. In this case, the DBMS assumes that transactions do not perform operations that violate the serializability criteria. However, the most widely used one is the MVCC. It creates a version of a tuple in the DB when a transaction updates it. On maintaining multiple versions of a tuple, it allows transactions to be completed even if another transaction updates the same tuple. It also allows long-term (and read-only) transactions not to block writers.

Another essential feature of a NewSQL DBMS is to provide recovery mechanisms in critical cases. Fault tolerance is the ability of the system to handle a structural failure. For traditional RDBs, the primary concern for fault tolerance is to ensure that no updates were missed [Mohan *et al*., 1992]. Unlike them, the NewSQL DBs must minimize downtime since modern applications are online all the time.

Besides, the NewSQL DBMSs use the relational data model and SQL as their primary language. They are built for distributed execution and optimized for multi-node environments. As the data are distributed over the cluster, one of the key problems of these architectures is the distributed transactions. So, organizing the data over the cluster in a way that reduces the number of distributed transactions is an important issue.

4 Heat Graph

In this paper, similar to Pavlo and Aslett [2016], and Curino *et al*. [2010], we consider a heat graph as a resource to represent the current workload of the database. A heat graph is a graph where each vertex has a weight. Each vertex on the graph represents an accessed tuple, and the weight is the number of times that the tuple is accessed. An edge between two vertexes v_1 and v_2 represents that the two vertexes are accessed together by a transaction.

In fact, each transaction touches a series of tuples. The tuples are mapped to vertexes, and all of them are connected by edges to signalize the relationship between them promoted by the transaction. The main goal of our heat graph is to represent each tuple with an *id* (primary key of the tuple) as well as the columns accessed by the transaction. It leads to multiple vertexes mapping the same tuple, each vertex representing a tuple with a specific projection. This reasoning let the partitioning algorithm handle the tuple in a more finned grain, allowing the creation of hybrid partitioning schemes. The definitions related to our heat graph are in the following.

Definition 1. Heat Graph. *A heat graph is defined by a tuple* $\mathcal{G} =< \mathcal{V}, \mathcal{E} >$*, where* $\mathcal{V}$ *is a set of vertex and* $\mathcal{E}$ *a set of edges.* ◇

Definition 2. Vertex. *A vertex* $v \in \mathcal{V}$ *is a tuple* $v = < v_{val}, v_{table}, Atts, \omega >$ *that represents an accessed tuple from the relational data model, where* v_{val} *is the vertex id (the rowid or the concatenation of the primary key values of the tuple),* v_{table} *is the table name,* $Atts$ *is the set of attributes form* v_{table} *accessed by the transaction, and* ω *is the number of times that the tuple with the projection was accessed, i.e., the heat of the vertex.* ◇

Definition 3. Edge. *An Edge* $e \in \mathcal{E}$ *is a tuple* $a =< v_1, v_2, w >$*, where* $v_1 \in \mathcal{V}$ *and* $v_2 \in \mathcal{V}$ *are two distinct vertexes, and* w *is the number of times that* v_1 *and* v_2 *are accessed together, i.e., the weight of the edge.* ◇

The heat graph stores the current workload of the system, since it maps each accessed tuple and the transaction that accesses the data. Each vertex represents a tuple (v_{val}) accessed with a specific projection ($Atts$), and every time the tuple is accessed using the $Atts$, the ω is incremented. Notice that if a tuple t_1 is accessed by one transaction with a projection π_1 and by another transaction with projection π_2 ($\pi_1 \neq \pi_2$), two different vertexes are generated: one for t_1 with $Attrs = \pi_1$ and another one for t_2 with $Attrs = \pi_2$. These two vertexes, which represent the same tuple accessed with different projections, are connected by an edge with no weight ($w = 0$).

Figure 3 (A) shows a subset of tuples from two tables: *Movies* and *Directors*. Figure 3(B), in turn, describes three different SQL transactions executed over these tables. Figure 4 shows the steps that generate the heat graph for the three transactions of Figure 3(B).

Suppose that the transactions I and II of Figure 3 are executed in parallel. When the first instruction of transaction I (t_1) and transaction II (t_2) are executed, two vertexes are added to the heat graph (Figure 4(A): blue for t_1 and red for t_2). The vertexes are labeled with information about the two

(A)

Table Movies			
id	name	director	year
1	Psyco	1	1960
11	The Godfather	2	1972
111	Patton	2	1970

Table Directors		
id	name	awards
1	Alfred Hitchcok	31
2	Francis Coppola	50

(B)

(I)
```
BEGIN
    UPDATE movies SET year = 1960 WHERE name = 'Psyco';
    SELECTname, year FROM movies;
COMMIT;
```

(II)
```
BEGIN
    SELECT id FROM directors WHERE name = 'Hitchcock'
    UPDATE movies SET director = 2 WHERE name = 'Psyco';
COMMIT;
```

(III)
```
BEGIN
    SELECT f.name, f.year, d.name FROM movie f JOIN
        director d ON f.director = d.id;
COMMIT;
```

Figure 3. (A) subset of tables Movies and Director; (B) three SQL transactions.

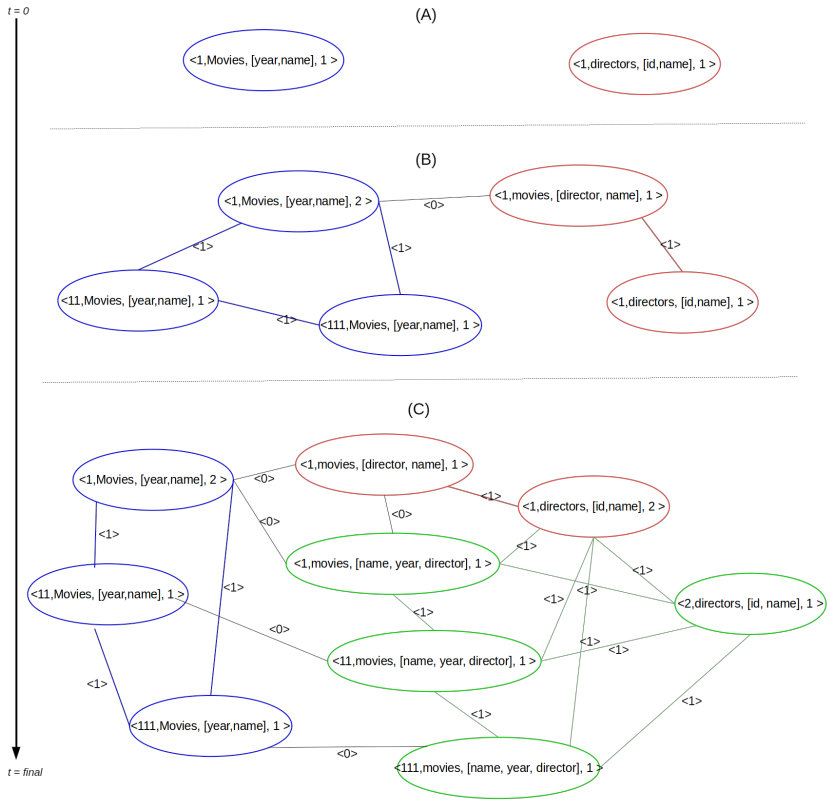

Figure 4. A heat graph for the running example of Figure 3.

transactions and their heats are set to 1. Notice that the two vertexes are not related since t_1 and t_2 touch different tuples at this moment.

When the second instructions of transactions I and II are executed, three new vertexes are added to the graph. They represent the three new tuples accessed by the transactions (two blue ones from t_1, and one red from t_2). Now, edges are added to the graph connecting all vertexes that were accessed by the same transaction (represented with the same color). Figure 4(B) presents the state of the graph after the complete executions of transactions (I) e (II). The vertex $< 1,' Movies', ['year',\ 'name'], 1 >$ (Figure 4(A)) has their weight updated to 2 in Figure 4(B) since the second instruction of the transaction (II) touches the tuple again with the same projection. Also, tuple 1 of the *Movies* table is accessed by the two transactions with different projections ([year, name] and [director, name]) creating two different vertexes in the graph connected by an edge with weight 0. At last, Figure 4(C) shows the heat graph after the execution of transaction (III).

5 Graph Partitioning

This section presents first the problem of graph partitioning, then a perfect solution based on linear programming, and at least a heuristic-based solution. Considering readers unfamiliar with linear programming, column generation, and/or the branch-and-price method, we decided to write a soft introduction in Section 5.2.1. The exact algorithm finds the optimal solution by using a branch-and-price method. All the codes are available in the GitHub[1]

5.1 Problem Definition

As stated before, our heat graph represents an abstraction of how data is accessed. Given a heat graph, we can partition it to represent where data are physically stored on the sites. Each partition has an identifier that is mapped directly to a physical data partition in a NewSQL DBMS. The heat graph is independent of the replication management, which is controlled by the NewSQL DBMS. In the following, we define a partition and a partitioning scheme.

[1] `https://github.com/gschreiner/graphPartitioning`

Definition 4. ***Partition***. *A partition is a tuple $\rho =< \mu, \Delta >$, where μ is a unique id for the partition, Δ is a subset of vertexes $\{v_1, v_2 \dots v_i\}$ $\forall i : v_i \in \mathcal{V}$, and i is the number of partition vertexes.* $\diamond$

Definition 5. ***Partitioning Scheme***. *A partitioning scheme is a tuple $\mathcal{S} =< \mathcal{G}, P >$, where $\mathcal{G}$ is a heat graph and P is a set of partitions $P = \{\rho_1, \dots, \rho_k\}$, where $(\rho_1 \bigcup \dots \bigcup \rho_k) = \mathcal{V}$ and $\forall\, i \neq j$ $(\rho_i \bigcap \rho_j = \emptyset)$.* $\diamond$

Each partition ρ has two basic edge types: (*i*) intra-partition, and (*ii*) inter-partition. An intra-partition edge connects two vertexes v_1 and v_2 that are in the same partition p_j ($v_1 \in p_j$ and $v_2 \in p_j$). In turn, an inter-partition edge connects two vertexes v_1 and v_2 from different partitions ($v_1 \in p_j$ and $v_2 \notin p_j$).

In a distributed system, it is essential to keep the system well-partitioned. On considering a NewSQL DBMS architecture, which is usually a shared-nothing one, data need to be partitioned in a balanced way because an unbalanced partitioning can lead to a latency rise when accessing data stored in a specific site. Thus, we consider Equation 1 to balance the system between the sites.

$$\forall i \in \{1 \dots k\} : \sum_{j<|p_i|}^{j=0} w(v_j) \leq (1+\alpha).(\frac{\sum_{l<|V|}^{l=0} w(v_l)}{k}) \tag{1}$$

Equation 1 states that the sum of the vertexes weights $w(v_j)$ for each partition $\rho_k \in P$ needs to be less or equal to the average workload per site. We obtain this average by summing the weights of all the vertexes v_l in the vertex set V and dividing by the number of partitions k. We also consider an unbalance factor α. When $\alpha = 0$, the system needs to be perfect balanced, otherwise α represent a perceptual ($0 \leq \alpha < 1$) for acceptable unbalancing, *i.e.*, if $\alpha = 0.1$ we accept 10% of unbalancing.

When the system needs to be re-partitioned, a new partitioning scheme is generated. A new partitioning scheme is a set P' of partitions where one vertex v_i migrates from one partition to another. Definition 6 formalizes this partitioning. Each new partition scheme to be considered as a valid partitioning scheme needs to be balanced according to Equation 1.

Definition 6. ***New Partitioning Scheme***. *A new partitioning scheme is a tuple $\mathcal{S}' =< \mathcal{G}, P' >$, where $\mathcal{G}$ is a heat graph and P' a new set of partitions P', where $\exists\, \rho' \in P'$ such that a $v_i \in \rho_l$, $\rho' = \rho' \bigcup v_i \bigwedge \rho_l = \rho_l / v_i$.* $\diamond$

Since it is desirable that a NewSQL DBMS avoids distributed transactions, we need to minimize the weight of the inter-partition edges and keep the system well-balanced. Equation 2 defines an objective function for minimizing inter-partition edges. Given k the number of partitions in the system, and $\xi = \{e_1, \dots, e_j\}$ the set of inter-partition edges, we cut (partition) the graph considering that the sum of inter-partition edge weights $w(e_i)$ needs to be closest to 0 as possible. If the sum is equal to 0, we achieve the perfect case and do not have any distributed transaction. Since it is a rare case, we tend to minimize the weighted sum by positioning the lower weight edgeS in the inter-partition area.

$$\xi \subseteq \mathcal{E},\ min(\sum_{i<|\xi|}^{i=0} w(e_i)) \tag{2}$$

Figure 5 shows a partitioned graph (partitioning scheme) based on the heat graph of Figure 4 (C) with modified weights. The graph has two partitions (P_1, P_2), and the red line shows the division (cut) on the graph. Each edge touched by the red line is considered an inter-partition edge. The graph of Figure 5 is unbalanced since $w(p1) = 24$ and $w(P_2) = 20$ and we have 12 distributed transactions (sum of inter-partition edge weights). An unbalanced graph can be acceptable depending of the α value (Eq. 1). If α is equal to 0.1 (10%) the system need to be re-partitioned since the difference between the partition loads is 4 ($w(P_1) - w(P_2)$) and the unbalancing allowed by α is 2. On increasing the α value to 0.2 (20%) the system remains unbalanced but in acceptable way considering that, now, α allows an unbalance of 4. Notice that $\alpha = 0.2$ solves the unbalancing problem, but we still have the inter-partition edges problem.

Figure 6 shows a new partitioning scheme based on Figure 5. On considering Equation 2, we reallocate two vertexes (the green ones) to two different partitions. Notice that, in the new partitioning scheme, the partitions are well-balanced and the number of distributed transactions decreased to 2 (yellow edge).

This section shows the principles used by our approach to minimize as much as possible the number of distributed transactions maintaining the system balanced. Our approach considers hybrid data partitioning, *i.e.*, based on the graph, we split data over the different sites (partitions) using vertical and horizontal data partitioning techniques. Critical edges (the inter-partition ones) with weight 0 are stored physically through a vertical partition. Edges with weight $w > 0$ are stored horizontally. Section 5.3 details our partitioning programming model[2].

5.2 Graph Partition - Analitical Solution

Linear programming consists of methods for solving optimization problems with restrictions in which the objective functions are linear [Gass, 2003]. Given the heat graph partition problem and Equations 3 and 4 to minimize the number of distributed transactions and maintain balanced system loads, it is possible to find the optimal partitioning for a given graph through linear programming methods using a solver. In the following, we present a linear programming model and the methods used to create the optimal partitioning solution. Initially, the column generation and branch-and-price methods applied in the approach are briefly presented. In sequence, the proposed model for the problem is presented.

5.2.1 Remarks on Column Generation and Branch-and-Price

Column generation is a method to solve mathematical linear programming problems with an exponential number of variables [Nash, 2013]. It is based on Dantzig-Wolfe decompositions [Dantzig and Wolfe, 1960]. For these problems, it is

[2]We consider here a model as a linear programming model/problem.

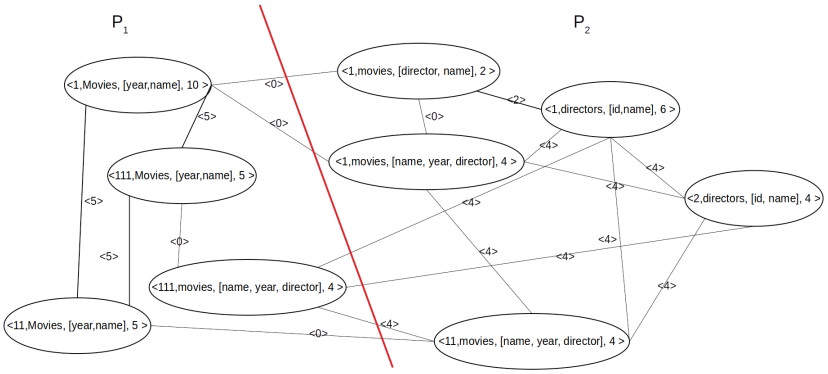

Figure 5. An unbalanced partitioning of the heat graph of Figure 4(C).

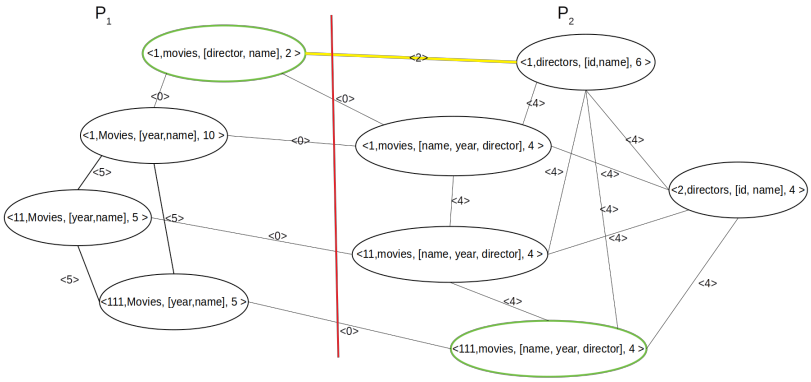

Figure 6. Re-partitioning of the heat graph of Figure 5 re-partitioned.

impossible to generate and store all variables so that column generation can solve larger instances for a model with this feature. In this context, each variable generated is a column for linear programming problems.

In the column generation method, the linear problem is called the *Master Problem (MP)*. It is necessary to formalize this problem and to identify which variables are exponentially numbered. Due to this number, a *Reduced Master Problem (RMP)* is specified and created with a small set of variables. This set should be enough to ensure a feasible non-optimal solution. Also, an *Auxiliary Problem (AP)* related to the identified variables should be defined. This problem uses the dual variable values obtained in the solving of the RMP. This problem finds a promising variable to enter into the column generation. If there is such a variable, it is said that a variable with a negative reduced cost is found [Desrosiers and Lubbecke, 2005]. We named these variables here as z. In this context, z_t is a variable from z at the index t in Section 5.3.

Figure 7 shows the generic column generation algorithm. It starts by creating the initial columns for the RMP. Then, the problem is solved and the dual values are obtained. These values are used by the AP to find a new promising variable (column). If such a variable exists, it is added into the RMP. If there is no such variable, then the column generation reached the optimal solution.

The column generation algorithm requires that all variables be continuous. It is an important requirement because the column generation algorithm relies on the AP to get new promising variables, and this problem uses the dual variable values. In order to attend this requirement when dealing with integer

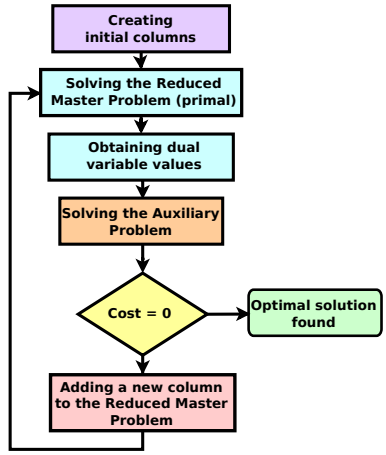

Figure 7. Column generation as an iterative process.

(or mixed-integer) linear programming, the integer variables of the MP are relaxed, and they are turned to continuous variables. With this relaxing approach, a column generation solution does not solve the integer (or mixed-integer) problem. So, another method is necessary to find the optimal solution in the integer solution space. It is called *branch-and-price*.

One key definition in the branch-and-price method is the concept of nodes. A node can be described as a part of the search space to explore. Each node deals with a particular search space of the problem by finding the continuous optimal solution (considering continuous variables) through the column generation procedure. In order to ensure the limits of such search space, each node has a set of constraints. The search for the best integer solution continues if there is a promising node with a continuous solution.

On considering a minimization problem, we can use Algorithm 1 to understand a branch-and-price method. The initial variables are defined to ensure a starting feasible solution (line 2), considering the RMP as a node. Then, a heap-min that ranks nodes is defined (line 3), and the initial branch-and-price node is stored (lines 4 and 5). Here, we are using the objective value as a ranking score, but this strategy could be different. The *upperbound* variable stores the objective value from the best integer solution found. At each iteration (lines 8 to 19), the branch-and-price node with the lowest objective value is obtained from the heap (line 9). If the solution of this node has a worse objective value than the *upperbound* value, its sibling nodes will not find a better solution, and the branch-and-price stops (lines 10, 11, and 12). If the search does not stop, the selected node is submitted to the column generation procedure.

Algorithm 1 A branch-and-price method for a minimization problem

1: create the *reduced master problem* **RMP** with initial variables
2: **heap** ← **new** *heap-min()* ▷ constructing the starting node of branch-and-price
3: **initialNode** ← ($objectiveValue$, $constraints = \{\}$)
4: **heap**.*push*(**initialNode**)
5: **upperBound** ← ∞
6: **stop** ← **false**
7: **while (not stop) do**
8: **data** ← **heap.***pop()*
9: **if data.***objectiveValue* > **upperBound then stop** ← **true**
10: **end if**
11: **if not stop then columnGeneration (RM, data, upperBound)**
12: **end if**
13: **if heap.***empty()* **then stop** ← **true**
14: **end if**
15: **end while**

During the branch-and-price, the column generation procedure starts adding the branch-and-price node constraints to the *RMP* (search space limits). At each iteration of the column generation, a new variable is added to the model by using the *Auxiliary Problem (AP)*. During the selection of a new variable, the *RMP* model is solved, and the values for the primal variables, the dual variables, and the objective value are stored. Then the AP is solved, and if a new variable is found, it is inserted into the *RMP* model. When no variable is found, the search stops. If the solution found has an integer objective value, the *upperbound* variable can be updated (if its value is better than the one stored in the *upperbound* variable). If an integer solution is not found, two branch-and-price nodes are created. The variable that has the largest fraction value has its value fixed in 0 for a node and 1 for the other node, dividing the search space into two parts to be addressed for future calls of the column generation procedure. These nodes are inserted into the heap-min.

5.3 The Column Generation Model

Model (3) in the following shows the graph partitioning problem (MP) for an undirected graph $G = (V, E)$, where V is the set of vertexes and E is the set of edges. This problem considers all subsets of V as possible clusters for a graph partition. So, consider that $T = 2^V$, K is the expected number of clusters, $c : E \rightarrow \mathbb{R}_*^+$ is a function that maps each edge cost, and $w : V \rightarrow$ is a function that maps the vertex cost. On considering the average cost of vertexes by cluster, $\alpha \geq 0$ is a fraction in which a cluster can exceed the "vertex cost balance". We also denote $a_{vt} = 1$ to represent that a vertex v belongs to a cluster t, and $a_{vt} = 0$ to represent that a vertex v does not belong to a cluster t.

Master Problem

$$\textbf{min} \sum_{\{u,v\} \in E} c(\{u,v\}) y_{\{u,v\}} + M \cdot h \tag{3a}$$

$$\textbf{subject to:} \sum_{t \in T} a_{vt} z_t = 1, \forall v \in V \tag{3b}$$

$$y_{\{u,v\}} \geq a_{ut}(1 - a_{vt}) z_t + a_{vt}(1 - a_{ut}) z_t, \forall \{u,v\} \in E, \forall t \in T \tag{3c}$$

$$\sum_{v \in V} w(v) a_{vt} z_t \leq (1 + \alpha) \cdot \frac{\sum_{v \in V} w(v)}{K}, \forall t \in T \tag{3d}$$

$$h \geq K - \sum_{t \in T} z_t \tag{3e}$$

$$h \geq -K + \sum_{t \in T} z_t \tag{3f}$$

$$z_t \in \{0,1\}, \forall t \in T \tag{3g}$$

$$y_{\{u,v\}} \in \{0,1\}, \forall \{u,v\} \in E \tag{3h}$$

$$h \geq 0. \tag{3i}$$

There are three main types of decision variables. If the cluster t is selected for the optimal solution, $z_t = 1$; otherwise $z_t = 0$. $y_{\{u,v\}} = 1$ denotes an inter-edge connecting two clusters by linking the vertexes u and v; otherwise $y_{\{u,v\}} = 0$. The variable $h = \left| \sum_{t \in T} z_t - k \right|$ denotes the absolute difference between the number of clusters expected and obtained. Such a variable is used to accept solutions with the number of clusters different from K during the branch-and-price procedure. M or big M is a large number used to assign 0 for the variable h.

The objective function in Equation $(3a)$ aims to find solutions for the graph partition with the minimal cost of inter-edges with a number of clusters equals to K. The constraint at inequation $(3b)$ requires that each vertex belongs to a single cluster in the optimal solution. In $(3c)$, the constraint ensures that $y_{\{u,v\}} = 1$ if $\{u,v\}$ connects vertexes from different clusters in the optimal solution, considering the objective function minimization. In turn, $(3d)$ ensures that each cluster t is restricted to the "accepted balance of vertex costs". Inequations $(3e)$ and $(3f)$ defines the h has as lowerbound the absolute difference between the expected and obtained number of clusters. Finally, in $(3g)$, $(3h)$, and $(3i)$, the bounds of the decision variables are defined.

Model (4) in the following shows the AP for the column generation. This model is based on the pricing problem: the best new z variable is selected to compose the reduced master problem. In this model, the values of $\lambda^{(i)}$ corresponds to the dual value of each constraint (i) in Model (3). The decision variables $a_v \in \{0,1\}$ represent if a vertex v belongs to the variable z for the master problem (a new column for the

reduced master problem). The constraints in $(4b)$, $(4c)$, $(4d)$, and $(4e)$ ensure that $y_{\{u,v\}} = 1$ if u and v are not in the selected cluster. In $(4f)$, $(4g)$, $(4h)$, $(4i)$, $(4j)$, $(4k)$, and $(4l)$, the bounds of decision variables are defined.

Auxiliary Problem

$$\min \sum_{\{u,v\}\in E} c(\{u,v\})y_{\{u,v\}} - \sum_{v\in V} a_v\lambda_v^{(1b)} + \sum_{\{u,v\}\in E} y_{\{u,v\}}\lambda_{\{u,v\}}^{(1c)} - \sum_{v\in V} w(v)a_v\lambda^{(1d)} - \lambda^{(1e)} + \lambda^{(1f)} \quad (4a)$$

$$\text{subject to: } y_{\{u,v\}} \geq a_u - a_v, \forall\{u,v\} \in E \quad (4b)$$
$$y_{\{u,v\}} \geq a_v - a_u, \forall\{u,v\} \in E \quad (4c)$$
$$y_{\{u,v\}} \leq a_v + a_u, \forall\{u,v\} \in E \quad (4d)$$
$$y_{\{u,v\}} \leq 2 - a_v - a_u, \forall\{u,v\} \in E \quad (4e)$$
$$y_{\{u,v\}} \in \{0,1\}, \forall\{u,v\} \in E \quad (4f)$$
$$a_v \in \{0,1\}, \forall v \in V \quad (4g)$$
$$\lambda_v^{(1b)} \in \mathbb{R}, \forall v \in v \quad (4h)$$
$$\lambda_{\{u,v\}}^{(1c)} \geq 0, \forall\{u,v\} \in E \quad (4i)$$
$$\lambda^{(1d)} \leq 0 \quad (4j)$$
$$\lambda^{(1e)} \geq 0 \quad (4k)$$
$$\lambda^{(1f)} \geq 0. \quad (4l)$$

The exact algorithm to solve the problem proposed here uses the column generation models of this section in the branch-and-price method of Algorithm 1 (Section 5.2.1).

5.4 Proposed Solution

This section details our approach. We first describe our partitioning process and, in the following, we present a running example for the better sake of understanding of our solution.

5.4.1 Partitioning Process

Our proposed process is a hybrid data partitioning algorithm that aims at minimizing the number of distributed transactions keeping the system well-balanced. It receives the number k of partitions of the system and the heat graph $\mathcal{G}$ as input parameters, and generates, as output, a new partitioning scheme that respects Equation 1 and Equation 2 requirements.

Algorithm 2 presents the proposed steps for the data partitioning problem. On considering the problem in a NewSQL system, we have three principal cases: (*i*) the system is unbalanced, (*ii*) there is a high number of distributed transactions, and (*iii*) the system is not partitioned. In the following, we explain each one of these cases and their related algorithms.

First of all, we have to evaluate the system according to Equation 1. It means that the algorithm needs to find out the minimum and maximum workloads, as well as the system's total load (lines 3-5 of Algorithm 2). In order to identify a possible unbalance in the system, we calculate the expected average of workload by dividing the total workload by the number k of partitions (line 6). The unbalance is detected if the maximum workload ($maxPartition$) is more than the average increase by α or if the minimum workload is less than the average workload decreased from α (line 7). If an unbalance is detected, we invoke the function $partitionUnbalancedGraph$ that creates a new partitioning scheme re-balancing the system (Case 1 - Algorithm 3).

If the system is well-balanced, we verify the number of distributed transactions (Case 2 - line 10). As previously stated,

Algorithm 2 Partitioning

```
1:  procedure Partitioning(G (heat graph), k (site number),α (workload unbalancing factor), θ (distributed transaction percentage))
2:      G' ← G
3:      minPartition ← getMinWorkload(G'.partitions)
4:      maxPartition ← getMaxWorkload(G'.partitions)
5:      sumLoads ← sumWorkload(G'.partitions)
6:      avgWorkload ← (sumLoads/G'.k)
7:      if (maxPartition > ((1 + α) * avgWorkload)) OR (minPartition < ((1 − α) * avgWorkload)) then
8:          G' ← partitioningUnbalancedGraph(G')
9:      end if
10:     if G'.criticalEdgesLoad > θ * sumLoads then
11:         G' ← partitioningMinimumCutGraph(G')
12:     end if
13:     if G'.k = 1 then
14:         G' ← firstPartitioning(G', k)
15:     end if
16:     if G ≠ G' OR G'.k ≠ k then
17:         G'.k ← k
18:         G' ← partitioning(G', k, α)
19:     end if
        return G'                ▷ G' (new partitioning scheme)
20: end procedure
```

the distributed transactions are mapped to critical edges, *i.e.*, edges that connect vertexes from different partitions. We use θ as a parameter that defines a minimum acceptable number of distributed transactions. The size of θ depends on the total number of distributed transactions, and we use 2% as the default value. Depending on the application's requirements, the user can change this parameter to relax the number of allowed distributed transactions (increasing the number of acceptable inter-partition edges) or restrict them. If we set the parameter with 0, the system will always try to reduce the minimum number of distributed transactions. If an unacceptable number of distributed transactions is detected, the system calls the function $partitionMinimunCutGraph$ that tries to find a new partitioning scheme that either respects Equation 1 and Equation 2.

If Case 2 is not true, the test for Case 3 (lines 13 to 15) verifies if exists some partitioning scheme in the system. Notice that if the system has just one partition, Case 1 and Case 2 are false because it is well-balanced and does not have any distributed transactions since it is a single node. In general, NewSQL systems are distributed and partitioned by nature. Nevertheless, Case 3 is the worst case since we have to create all the partitioning schemes from scratch. If the system was not previously partitioned, we invoke the function $firstPartition$ that creates a partitioning scheme based on the k number of partitions.

At the end of Algorithm 2, we verify if the new partition scheme $\mathcal{G}'$ is different from the original partition scheme $\mathcal{G}$. If the $\mathcal{G}'$ is different from the original, we recursively call the partitioning function. This call is needed because Case 3 will probably generate a well-balanced partition scheme,

but probably with a high number of distributed transactions since we use a coarsening technique to reduce the size of the graph (more details in Algorithm 5). Also, this recursive call guarantees that Case 1 and Case 2 be respected.

As mentioned before, Case 1 occurs when the system is unbalancing. In order to detect this problem, we maintain the total number of accesses for each partition and periodically test against Equation 1 (Algorithm 2). The algorithm for the function $partitioningUnbalancedGraph$ generates a new partitioning scheme that balance the workload (Algorithm 3). The main idea is to migrate tuples from the most overloaded partitions to the underloaded ones.

Algorithm 3 initially gets the most underloaded and overloaded partitions (lines 2 and 3). It is possible that we have the same workload for more than one partition (underloaded or overloaded). In this case, we return the partition with the smallest identifier. As on the previous algorithm, the average workload is obtained by dividing the total workload by the number of partitions (line 4). We keep searching for hot tuples in the overloaded partition and migrating them to the underloaded partition until the overloaded partition respects Equation 1 (line 5 to 7). In order to get the candidate tuple for migration between the partitions, we use the function $mostProeminentTuple$.

The function $mostProeminentTuple$ gets a tuple (called *candidate*) that is eaten the hottest vertex with the minimal degree (minimal sum of vertexes weight edges). In some cases, we explore the neighborhood of the vertex. On considering our heat graph, this approach can lead to a minimal critical edge. After a candidate is selected, we migrate the tuple to the underloaded partition. In that migration process, we need to consider migrating some of the neighborhood (adjacent vertexes) of the candidate allocating the minimal edge in the critical area (area between partitions). The exploration of a vertex neighborhood is accomplished by using a factor of hopes whose default value is 2, and the process considers that the target partition will not be overloaded after the migration. When the candidate (and eventually their neighborhood) is migrated to the new partition, we re-evaluate Equation 1 and migrate more vertexes until the equation is satisfied.

After the two partitions are well-balanced, the algorithm tests if the unbalance of the system is solved (line 9 to 14). As previously, we select the most overloaded and underloaded partitions, and apply the balance equation. If an unbalance is detected, the algorithm calls recursively the function $partitionUnbalancedGraph$. This process recursively runs until no more unbalanced partitions are found.

The second case (Case 2) occurs when the system is well-balanced, but we have a high number of distributed transactions (Algorithm 2 at line 10). In order to minimize this number, we analyze the critical edges. We select all the critical edges in a decreasing order by the weight. For each edge, we analyze and create new partitioning schemes called *candidates*. The candidates need to be valid for the Equations 1 and 2. The Algorithm 4 presents a pseudo-code for the function $partitioningMinimumCutGraph$, which deals with Case 2.

The algorithm iterate over all the critical edges (lines 2 and 3) investigating if is possible to migrate one of the vertexes in order to minimize the total weight of the critical edges. It selects the hottest edge (the bigger weight), stores it in ce', and tries two possibilities of migration: (*i*) migrate the origin of ce' to the partition of the target; and (*ii*) migrate the target of ce' to the partition of the origin. For each of them, we analyze the migration of the vertexes and their neighborhood in order to find the minimal cut for the migration. The neighborhood is explored by also considering a number of hopes, whose default is 2. The exploration tries to get the minimum cut (minimal edge weight) without disrespecting the Equation 1.

When the function $explorePossibility$ finds a valid candidate, it returns the weight of the new critical edge. If no candidate is available, the function returns -1. We execute the function $explorePossibility$ for each possibility (lines 5 and 6), and compare them to find out the best candidate. The algorithm first checks if the two values are valid options, if both are equal to -1, no valid option is available (lines 7 to 9). If at least one of the options is valid, then we test if the migration option of the target is valid and less than the origin. If so, the migration is executed (line 12). Otherwise (line 13 to 19), the algorithm checks if the origin option is valid and migrates it. The function $explorePossibilitiesReplication\mathcal{G}$ (line 8), in turn, explores the possibility of replicating one or more tuples if all other options are not possible. The main idea is to analyze the replication of one of the tuples (and, if necessary, its neighborhood) from the critical edge to reduce the number of distributed transactions. In order to use the replication option the new $\mathcal{G}'$ need to satisfy both equations (Equation 2 and Equation 1). This function is used as the ultimate alternative since it increases the size of the database by inserting a replica of a tuple (and its neighborhood) to eliminate the critical edge.

Case 3 considers a system without partitioning, *i.e.*, we have only one partition. Notice that, in this case, the system is well-balanced and does not exist any distributed transactions since all sites have all the data (the system runs in a single node mode). When the system has no partitions or just one, we need to generate an initial partitioning scheme. When a new partitioning is generated, it is submitted recursively to the other cases. The Algorithm 5 shows the pseudo-code for the function $firstPartitioning$, which deals with this case.

Algorithm 5 first initializes the set of partitions. It creates an initial graph with k partitions (line 2). The version of our heat graph uses vertical partitioning, which leads to a bigger graph since one tuple can be mapped to more than one vertex. The size of the graph size affects the performance of the algorithm since each vertex is explored to create the partition scheme. Given this problem, we use a coarsening technique to reduce the number of vertexes in the graph (line 3). The $coarseningGraph$ function shrinks the graph by combining vertexes linked by edges with no weight (vertexes that represent the same tuple). In the following, we allocate each of the hottest tuples in one partition (lines 4 to 7). When a hot tuple is allocated, the function $allocateVertex$ selects the partition more suitable for the vertex and verifies each edge of the vertex keeping hot edges in the same partition to avoid distributed transactions.

After the first allocation of the hottest vertexes, each one of the other vertexes is allocated to the correct partition. One by one, the algorithm verifies, based on the edge weights, which partition is the best option for allocating the vertex

Algorithm 3 Partitioning Unbalanced Graph

procedure PartitioningUnbalanced($\mathcal{G}$ (Heat Graph); $\mathcal{S}$ (Partition Scheme), k (Site Number))
  $underloadedPartition \leftarrow mostUnloadedPartition(k)$
  $overloadedPartition \leftarrow mostOverloadedPartition(k)$
  $avgWorkload \leftarrow (\mathcal{G}'.totalLoads/\mathcal{G}.k)$
  while $(\mathcal{G}'.partition[underloadedPartition].load <= ((1+\alpha) * avgWorkload)$ **do**
    $n' \leftarrow mostProeminentTuple(\mathcal{G}'.partition[overloadedPartition])$
    $realocateTuple(\mathcal{G}'.partition[underloadedPartition], n')$
  end while
  $overloadedPartition \leftarrow mostOverloadedPartition(k)$
  $underloadedPartition \leftarrow mostUnloadedPartition(k)$
  if $(\mathcal{G}'.partition[overloadedPartition].load > ((1+\alpha) * avgWorkload)) \; OR$
$(\mathcal{G}'.partition[underloadedPartition].load < ((1-\alpha) * avgWorkload))$ **then**
    $\mathcal{G}' \leftarrow partitionUnbalancedGraph(\mathcal{G}')$
  end if
  return $\mathcal{G}'$ ▷ $\mathcal{G}'$ (new partitioning scheme)
end procedure

Algorithm 4 Partitioning Minimum Cut Graph

procedure PartitioningMinCut($\mathcal{G}$ (heat graph); $\mathcal{S}$ (partitioning scheme), k (site number), β (acceptable depth search))
  $criticalEdges \leftarrow \mathcal{G}.criticalEdges$
  for $ce' \in criticalEdges)$ **do**
    $ce' \leftarrow hottestEdge(citicalEdges)$
    $chargeTestOrigin \leftarrow explorePossibility(ce'.origin, ce'.target.partition)$
    $chargeTestTarget \leftarrow explorePossibility(ce'.target, ce'.origin.partition)$
    if $chargeTestOrigin < 0 \; AND \; chargeTestTarget < 0$ **then**
      $\mathcal{G}' \leftarrow explorePossibilitiesReplication\mathcal{G}$ **return** $\mathcal{G}'$
    end if
    if $chargeTestOrigin > chargeTestTarget \; AND \; chargeTestTarget >= 0$ **then**
      $\mathcal{G}' \leftarrow migrateTuple(ce'.target, ce'.origin.partition)$
    else
      if $chargeTestOrigin >= 0$ **then**
        $\mathcal{G}' \leftarrow migrateTuple(ce'.origin, ce'.target.partition)$
      else
        $\mathcal{G}' \leftarrow \mathcal{G}$
      end if
    end if
  end forreturn $\mathcal{G}'$
end procedure

Algorithm 5 First Partitioning

```
1:  procedure DefaultPartitioning(G (heat graph); S (partitioning scheme), k (site number))
2:      G' ← initializePartitions(G, k)
3:      G' ← coarseningGraph(G')
4:      for partition' ∈ G' do
5:          t' ← hottestTuple(G')
6:          G' ← allocateVertex(G', t')
7:      end for
8:      for t' ∈ hottestTupleNotVisited(G') do
9:          G' ← allocateVertex(G', t')
10:     end for
11:     G' ← uncoarseningGraph(G')
12:     G' ← partitioning(G')
        return G'
13: end procedure
```

(lines 8 to 10). Like the previous ones, this process considers the equations to keep the system well-balanced and with a minimal number of distributed transactions at the same time.

In the following, the graph is uncoarsed to split the vertexes that were previously combined (line 11). This process is time consuming since each vertex needs to be explored and eventually split. After that, the algorithm calls the partitioning function recursively to submit the new data partitioning scheme to the main process. Despite this worst performance, this Case 3 algorithm does not represent a problem for our approach, since it is not common, being called usually one time.

5.5 Running Example

For sake of understanding, we perform a step by step of our approach, and we compare our resulting partitioning scheme with the optimal solution. For the optimal solution, however, it is not possible to show a step by step since the solver explores the graph using a traversing technique to validate all possibilities and raise one of the optimal partitioning schemes.

Figure 8 shows a heat graph for an initial workload for the IMDb database. The workload is very basic, *i.e.*, the graph is limited to one transaction for each tuple. As described previously, each vertex of the graph represents one accessed tuple, and edges connect two vertexes that are accessed together. Each edge has a weight that represents the number of times that the two vertexes are accessed simultaneously, and edges with no weight (0) represent the same tuple accessed with a different projection. In order to simplify the process, we run the algorithms with two partitions ($k = 2$) and $\alpha = 20\%$ (acceptable unbalance).

On considering our Algorithm 2, the first test is the verification of the workload balance of the system, since we consider the number of partitions of the system (1, as the system was not previously partitioned), and divide the total workload (12, the sum of the weight/heat of all vertexes - see Figure 8). The balancing equation returns a perfectly balanced system, as a system with just one partition is well-balanced. In the following, our algorithm tests the number of distributed transactions. As in the previous situation, since the system has only one partition, no distributed transactions are detected. Then, the Algorithm verifies if the partitioning scheme fits Case 3 (Algorithm 5).

Since the graph was not previously partitioned, the first task is the coarsening of the graph. The coarsening process reduces the number of vertexes of the graph by combining vertexes of the same accessed tuple connected through edges with weight 0. It generates a single vertex for the tuple with the sum of the heat of the previous vertexes. Figure 9(A) shows the coarsen graph for the graph shown in Figure 8. For example, the original graph has a vertex $< 5, movies, [d], 1 >$ connected to $< 5, movies, [y, n], 1 >$ by an edge with weight 0. After the coarsening process, these two vertexes are combined into a new one $< 5, movies, [d, y, n], 2 >$.

From this compact version of the graph, our approach starts the vertexes exploration to create a new partition scheme. Two partitions are created, considering $k = 2$. At this point, we put the hottest tuples in different partitions. In our example, all the vertexes have the same weight (2). So, we pick up the first k vertexes (by their identifiers) ($< 1, movies, [d, y, n], 2 >$ and $< 2, movies, [d, y, n], 2 >$) and allocate them in different partitions. These vertexes will be the basis for expanding the partitions.

One-by-one, the rest of the vertexes are explored and allocated to the partitions. A vertex v_i is considered a potential vertex to migrate to a partition p_j if p_j has more vertexes accessed together with v_i, considering that this migration will not overload the partition. Figure 9(B) shows the result of this step of our algorithm. Notice that the system is well-balanced and has four distributed transactions.

Next, we undo the coarsening process to expand the graph back to the full version without affecting the partitions. At this point, we call again our main Algorithm passing as a parameter the new partitioning scheme. As the process was executed over a compressed version of the graph, we probably generate a partitioning scheme with workload balance problems and/or distributed transactions that can need to be eliminated considering the complete heat graph. So, the process must go on.

Figure 10(A) shows the graph after the uncoarsening process. It is the input for a new call of our partitioning algorithm. We then repeat the process of analyzing the three cases. First, the algorithm verifies the graph workload balance. As the graph has two partitions, each one with a workload of 6, the graph is well-balanced. Next, we test the critical edges.

Critical edges are analyzed one by one by exploring different migrations in order to minimize the number of distributed transactions. This is the responsibility of Algorithm 4. A critical edge is an edge ϵ that has $\epsilon.weigth > 0$ and connects two vertexes from different partitions. In order to analyze the critical edges, we sort them all by their weight, and each edge is analyzed once. Giving the hottest edge, we consider two possibilities: (i) to migrate the origin vertex to the target partition; (ii) to migrate the target vertex to the origin partition.

When our algorithm tests the migration of a vertex v_i to another partition, it analyzes the neighborhood of v_i to find the best cut. For example, when migrating v_i to another partition, we can eliminate a critical edge and create a new one if v_i is connected to another vertex with a hot edge that is now placed between partitions. In order to overcome this problem, we analyze the neighborhood of v_i evaluating the best cut.

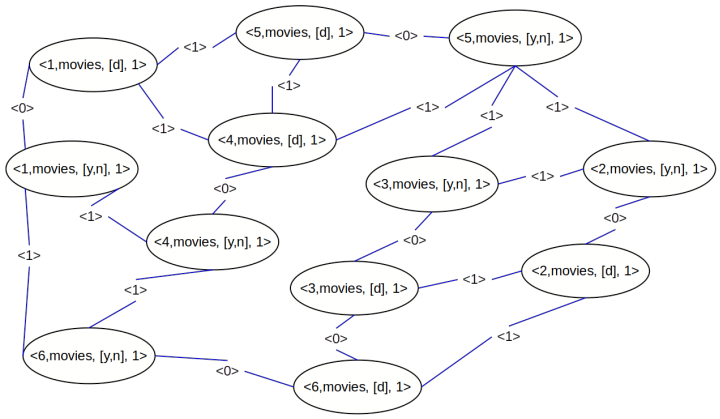

Figure 8. Initial heat graph for the IMDb workload

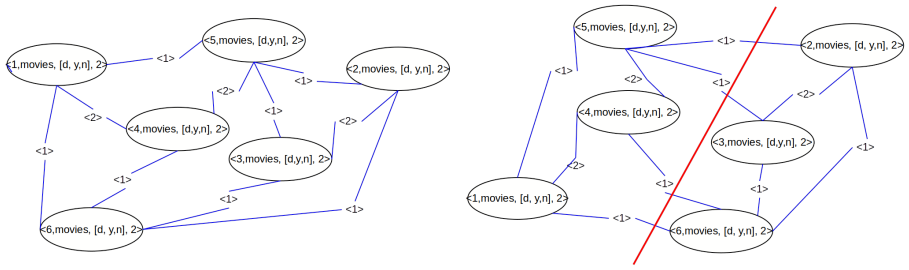

Figure 9. Coarsen Graph IMDb workload

The purpose here is to minimize the total weight of the critical edges. However, this process can lead to performance problems since many possibilities may be explored. In order to solve this issue, our algorithm is limited to β hops (the default is two), which limits the number of possibilities. After the algorithm explores the two migration options, it compares if the two options are valid. An option is valid if it respects Equation 1, and the new critical edge is less than the original.

We then select the best migration option, *i.e.*, the one with minimal weight, and migrates the vertex, as well as its neighborhood, if necessary. Given the graph of Figure 10(A), the first analyzed edge is the connection between the vertexes $< 5, movies, [y, n], 1 >$ and $< 2, movies, [y, n], 1 >$ with weight 1. If we migrate the vertex $< 2, movies, [y, n], 1 >$ to the other partition, the graph remains balanced (considering the unbalance factor α), but we still have one critical edge with the same weight. If we expand this option to the neighborhood of the vertex, we would have to migrate also the vertexes $< 2, movies, [y, n], 1 >$ and $< 3, movies, [y, n], 1 >$. It would eliminate the critical edge, but would generate an unbalanced graph (larger than α). The second option is to migrate the vertex $< 5, movies, [y, n], 1 >$ to the other partition, which reduces the critical edges. This is chosen option.

The second critical edge to be analyzed is the one that connects the vertexes $< 4, movies, [y, n], 1 >$ and $< 6, movies, [y, n], 1 >$ with weight 1, since the edge between $< 5, movies, [y, n], 1 >$ and $< 3, movies, [y, n], 1 >$ was solved with the previous migration. After analyzing the possible options, the algorithm decides to migrate the vertex $< 6, movies, [y, n], 1 >$. Figure 10(B) shows the result of the partitioning process.

Other partitioning options could be explored, and the results depend on the first partitioning scheme. On considering the optimal solution (Section 5.1), Figure 11 shows the generated partitioning scheme. Notice that the optimal solution has a better partition scheme, without distributed transactions, and our solution, instead, has only one. The difference between the approaches is the execution time. The optimal solution finds the best partitioning scheme exploring all the possibilities of the graph. Our solution is based on a more efficient algorithm that generates a partitioning scheme in less time. This trade-off between quality and execution time is discussed in the literature [Buluç *et al.*, 2016].

6 Experimental Evaluation

This section describes two experiments that evaluate the quality of our partitioning strategy. We compare ourselves with the partitions generated by the solver (optimal solution) and Clay (a state-of-the-art approach), considering different graph sizes. The graph partitioning problem does not have one possible solution. So, in order to compare two partitioning schemes, we evaluate the number of distributed transactions.

We use the IMDb data model and generate random queries over the schema. The queries have different filters and projections, and some of them use the join operation. Based on the execution of each operation over the data, we collect the heat graph and execute each one of the methods.

Since we focus on the graph quality, we previously collect and store a graph, and use it as input for each solution. Clay collects the graph in execution time, but we do not work this way. We just consider the Clay algorithm to generate the

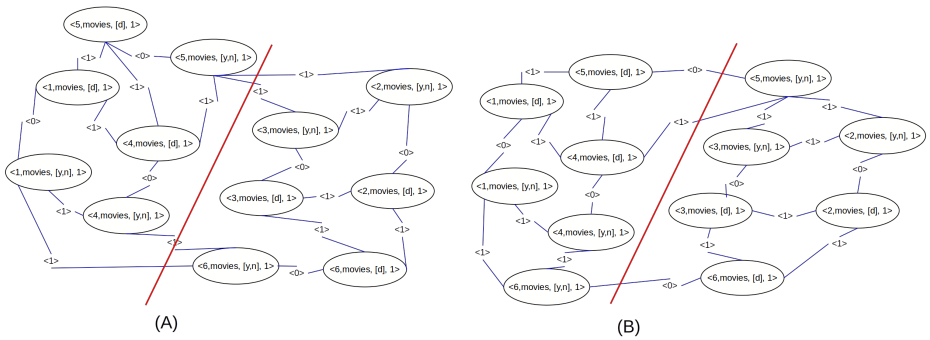

Figure 10. Graph after the execution of Case 3

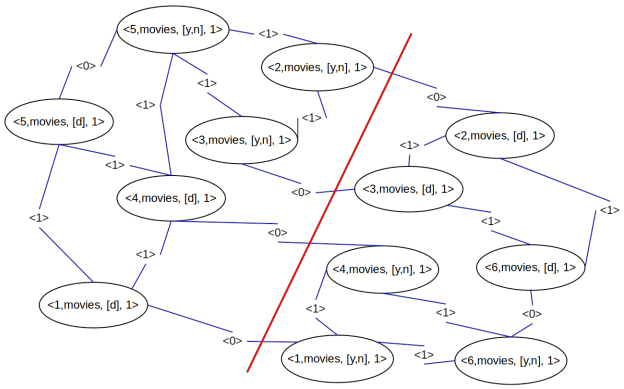

Figure 11. Partitioning scheme for the optimal solution

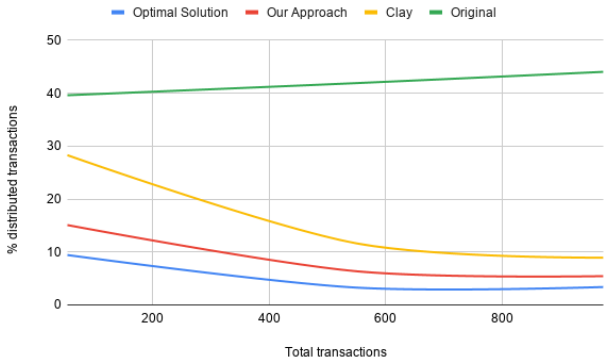

Figure 12. Partitioning scheme vs. Optimal solution

partitions.

We do not compare the different approaches in terms of execution time since the optimal solution explores the graph traversal for all the possibilities. This method is efficient to find out the best partitioning scheme, but it is inefficient in terms of time execution. So, we could not effectively test massive graphs in the comparison with the optimal solution, limiting the experiments to 100 vertexes (971 transactions), which take seven days to create the partitioning scheme.

6.1 Experiment One

Our first experiment compares the optimal solution with our approach and Clay. It consists of different graph sizes. For each size, we execute the three methods and compare the number of distributed transactions. Our graph represents the transactions executed over the IMDb DB. It holds seven tables (*name_basics*, *title_ratings*, *title_principals*, *title_episode*, *title_crew*, *title_basics*, *title_akas*) and the executed transactions are a mix of queries that combines different tables with join operations, as well as random filters and projections. The graph has three different sizes: (*i*) 15 vertexes and 53 edges; (*ii*) 50 vertexes and 551 edges; and (*iii*) 100 vertexes and 974 edges.

Figure 12 shows the results of the experiment. The graph combines the percentage of the distributed transactions with the total number of transactions. Each line of the graph represents one approach: (*i*) the optimal solution; (*ii*) our approach using $\alpha = 20\%$; (*iii*) the Clay approach, and; (*iv*) the percentage of distributed transactions without the use of any partitioning algorithm (original).

According to Figure 12, the percentage of distributed transactions of the original partition scheme without using a proper graph partitioning approach is around 40%. For the first size of the graph (53 edges), the number of distributed transactions variate from 28% for the Clay approach to 9% for the optimal solution. The dimension of the graph causes this variation and a high number of distributed transactions. With a reduced number of transactions, the algorithms have fewer options for reorganizing the data. We also notice that, as the size of the graph increases, less distributed transactions are detected since we decrease the probability of two tuples be used by the same transaction in the same way. The figure also shows that our approach is closer to the optimum (optimal solution) than Clay. The main reason is the graph itself. Our graph is bigger than the Clay graph as we consider hybrid data partitioning. This extended graph allows our approach to partition the graph in a more fine-grained way, which position different tuple projections in distinct partitions.

6.2 Experiment Two

Our second experiment considers the *Voter* benchmark. As previously stated, we collected the workload graph and use them as input for each method. The Voter is a benchmark based on the software used by a television talent show aired in Japan and Canada. Users call to vote for their favorite candidate. Upon receiving a call, the application invokes the transaction that updates the total number of votes of each participant. Votes cast by each user are stored in a DB and have a configurable upper limit. A separate transaction is periodically invoked to compute the total number of votes during the program.

This benchmark was developed to saturate the DB with small transactions updating a small number of records. The *Voter* DB is composed of three tables that hold data about

the candidates and the calling user. In addition, there are two *views* that are queried to update the status of the television program. As in the previous experiment, we compare the percentual of distributed transactions. The size of the graph variates from 5000 to 10000 edges. Giving the size of the graphs, it is impossible to run the solver as it would spend too much time to generate the optimal solution. So, we compare our approach only against Clay. Figure 13 shows the result of the experiment.

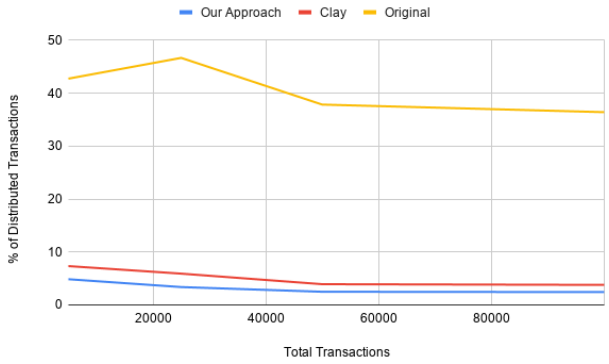

Figure 13. Comparison of distributed transactions using the Voter benchmark

The result shows that our approach generates a better partitioning quality if compared to Clay. As discussed before, the main reason is that our heat graph considers a hybrid data partitioning strategy that decreases the number of distributed transactions. As in the previous experiment, as the graph size increases the percentage of distributed transactions decreases. For all the observed graph sizes our approach got, on average, 2% less distributed transactions than Clay.

6.3 Results Discussion

The experiments conducted in this work demonstrate the effectiveness of our partitioning strategy when compared to both the optimal solution and the state-of-the-art Clay algorithm. In both scenarios (IMDb and Votter), our approach consistently produces partitioning schemes that result in a lower percentage of distributed transactions.

In the first experiment, which includes a comparison with the optimal solution, we observe that our strategy achieves results that are closer to the optimal than those produced by Clay. Clay reduces the percentage of distributed transactions to 28% in smaller graphs, while our approach brings it down to approximately 14%, approaching the 9% achieved by the optimal solution. As the graph size increases, the proportion of distributed transactions naturally decreases, which reflects the increased diversity in transaction access patterns and reduced overlap between tuples.

A key reason for the improved performance of our approach is its reliance on the heat graph. The capability of using the hybrid data partitioning allows a more fine-grained structure to reorganize the data. This advantage becomes more evident in larger and more complex workloads.

The second experiment shows that the average improvement of around 2% fewer distributed transactions demonstrates the scalability of our technique under high-throughput, transactional workloads, such as those modeled in the *Voter* benchmark.

Overall, the results validate the effectiveness of our approach in reducing distributed transactions, which is a critical factor in improving performance and scalability in distributed database systems.

Although it is important to note that the optimal solution does not apply to real scenarios, as it explores all the alternatives of the partition to find the optimal one. In contrast, the proposed heuristic solution is more lightweight and uses the number of hops to optimize the process of exploring new partition schemes. The heuristic solution does not explore the entire graph. However, it analyzes critical edges (distributed transactions) and, for each critical edge, explores a reorganization based on the neighborhoods of the node, limited by the number of hops (the distance between the node of the critical edge and its neighborhood, measured in terms of the number of edges). As we increase the number of hops, we explore more partition schemes but also increase the execution time. We use two as the standard number of hops based on empirical tests.

7 Conclusion

This paper presents a novel approach for data partitioning in the context of NewSQL DBs. It is based on an extended heat graph that represents the distributed DB transactions, as well as a *hybrid partitioning* strategy. This strategy considers two objective functions that partition the graph keeping well-balanced graph partitions (local transactions equally distributed) and minimizing inter-partition edges (distributed transactions).

We also present an optimal solution for the data partitioning using a solver to generate the optimal data partitioning for the objective functions. The solver solution is used as a baseline to show how close our approach is to the optimal situation. We also detail the high-level algorithms of our approach and provides a case study to illustrate their execution.

Additionally, we execute a set of experiments to evaluate the quality of our partitioning schemes. Besides the baseline, we also compare our results against the Clay state-of-the-art approach. The experiments show that our approach got a better partitioning, generating less distributed transactions. As the optimal solution explores all the possibilities of tuple migration to find an optimal scheme, its time consumption is prohibitive for massive graphs. So, we propose another experiment considering more massive graphs to compare only against Clay. Our results also outperform Clay, revealing that our approach is promising and confirming our hypothesis.

Future works include the evaluation of our approach considering other benchmarks and more extensive tests. Explore the partition technique in HTAP workloads. We also intend to evaluate our partitioning strategy in terms of latency and throughput for storing data in the Hybrid-VoltDB [Schreiner *et al.*, 2019]. Also, we intend to utilize the algorithm with live migration implemented within the new SQL system.

Declarations

Authors' Contributions

GAS and RS worked to the conception of this study. GAS and RS performed the experiments. GAS is the main contributor and writer of this manuscript. DD and RSM help write and revise the manuscrit. All authors read and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Availability of data and materials

The datasets (and/or softwares) generated and/or analysed during the current study will be made upon request.

References

Al-Kateb, M., Sinclair, P., Au, G., and Ballinger, C. (2016). Hybrid row-column partitioning in teradata. *Proc. VLDB Endow.*, 9(13):1353–1364. DOI: 10.14778/3007263.30072.

Amossen, R. R. (2010). Vertical partitioning of relational oltp databases using integer programming. In *ICDEW*, pages 93–98. IEEE. DOI: 10.1109/ICDEW.2010.5452739.

Arulraj, J., Pavlo, A., and Menon, P. (2016). Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 583–598, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2882903.2915231.

Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., and Schulz, C. (2016). Recent advances in graph partitioning. *Algorithm engineering*, pages 117–158. DOI: 10.1007/978-3-319-49487-6_4.

Cetintemel, U., Du, J., Kraska, T., and Madden, e. a. (2014). S-store: A streaming newsql system for big velocity applications. *Proc. VLDB Endow.*, 7(13). DOI: 10.14778/2733004.2733048.

Curino, C., Jones, E., Zhang, Y., and Madden, S. (2010). Schism: A workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2). DOI: 10.14778/1920841.1920853.

Dantzig, G. B. and Wolfe, P. (1960). Decomposition principle for linear programs. *Oper. Res.*, 8(1):101–111. DOI: 10.1287/opre.8.1.101.

Desrosiers, J. and Lubbecke, M. E. (2005). A Primer in Column Generation. In *Column Generation*, volume 3, pages 1–32. Springer-Verlag, New York. DOI: 10.1007/0-387-25486-2.

DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A. (1984). Implementation techniques for main memory database systems. *SIGMOD Rec.*, 14(2):1–8. DOI: 10.1145/971697.602261.

Elmore, A. J., Arora, V., Taft, R., Pavlo, A., Agrawal, D., and El Abbadi, A. (2015). Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *2015 ACM SIGMOD*, pages 299–313, New York, NY, USA. ACM. DOI: 10.1145/2723372.2723726.

Gass, S. I. (2003). *Linear programming: methods and applications*. Courier Corporation. DOI: 10.2307/2006141.

Grolinger, K., Higashino, W. A., Tiwari, A., and Capretz, M. A. (2013). Data management in cloud environments: Nosql and newsql data stores. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):22. DOI: 10.1186/2192-113X-2-22.

Kallman, R., Kimura, H., Natkins, Stonebraker, M., *et al*. (2008). H-store: a high-performance, distributed main memory transaction processing system. *VLDB*, 1(2). DOI: 10.14778/1454159.1454211.

Kumar, R., Gupta, N., Charu, S., and Jangir, S. K. (2014). Manage big data through newsql. In *National Conference on Innovation in Wireless Communication and Networking Technology–2014, Association with THE INSTITUTION OF ENGINEERS (INDIA)*, pages 1–5. DOI: 10.13140/2.1.3965.3768.

M. Tamer Özsu, P. V. a. (2011). *Principles of Distributed Database Systems, Third Edition*. Springer-Verlag New York, 3 edition. Book.

Ma, L., Arulraj, J., Zhao, S., Pavlo, A., Dulloor, S. R., Giardino, M. J., Parkhurst, J., Gardner, J. L., Doshi, K., and Zdonik, S. (2016). Larger-than-memory data management on modern storage hardware for in-memory oltp database systems. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, DaMoN '16, pages 9:1–9:7, New York, NY, USA. ACM. DOI: 10.1145/2933349.2933358.

Meehan, J., Tatbul, N., Zdonik, S., Aslantas, C., *et al*. (2015). S-store: Streaming meets transaction processing. *Proc. VLDB Endow.*, 8(13). DOI: 10.48550/arXiv.1503.01143.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. (1992). Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162. DOI: 10.1145/128765.128770.

Nash, S. G. (2013). *Encyclopedia of Operations Research and Management Science*. Springer US, Boston, MA. DOI: 10.1016/s0898-1221(97)90013-4.

Navathe, S., Ceri, S., Wiederhold, G., and Dou, J. (1984). Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)*, 9(4):680–710. DOI: 10.1145/1994.2209.

Pavlo, A. and Aslett, M. (2016). What's really new with newsql? *SIGMOD Rec.*, 45(2):45–55. DOI: 10.1145/3003665.3003674.

Pavlo, A., Curino, C., and Zdonik, S. (2012). Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *2012 ACM SIGMOD*, pages 61–72, New York, NY, USA. ACM. DOI: 10.1145/2213836.221384.

Pavlo, A., Jones, E. P. C., and Zdonik, S. (2011). On predictive modeling for optimizing transaction execution in parallel oltp systems. DOI: 10.14778/2078324.2078325.

Schreiner, G. A., Duarte, D., Dal Bianco, G., and Mello, R. d. S. (2019). A hybrid partitioning strategy for newsql databases: The voltdb case. In *Proceedings of the 21st International Conference on Information Integration and*

Web-Based Applications & Services, iiWAS2019, page 353–360, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/3366030.3366062.

Serafini, M., Mansour, E., Aboulnaga, A., Salem, K., Rafiq, T., and Minhas, U. F. (2014). Accordion: Elastic scalability for database systems supporting distributed transactions. *Proc. VLDB Endow.*, 7(12):1035–1046. DOI: 10.14778/2732977.2732979.

Serafini, M., Taft, R., Elmore, A. J., Pavlo, A., Aboulnaga, A., and Stonebraker, M. (2016). Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456. DOI: 10.14778/3025111.3025125.

Sharma, G. and Kaur, P. D. (2015). Architecting solutions for scalable databases in cloud. In *Proceedings of the Third International Symposium on Women in Computing and Informatics*, pages 469–476. ACM. DOI: 10.1145/2791405.2791432.

Stonebraker, M. (2012). New opportunities for new sql. *Commun. ACM*, 55(11). DOI: 10.1145/2366316.2366319.

Taft, R., Mansour, E., Serafini, M., Duggan, J., Elmore, A. J., Aboulnaga, A., Pavlo, A., and Stonebraker, M. (2014). E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3). DOI: 10.14778/2735508.2735514.

Valdes, J., Garcia-Molina, H., and Lipton, R. (1984). A massive memory machine. *IEEE Transactions on Computers*, 33(05):391–399. DOI: 10.1109/TC.1984.1676454.