A Hybrid Partitioning Strategy for NewSQL Databases: The VoltDB Case

Geomar A. Schreiner
PPGCC - Federal University of Santa Catarina
Florianópolis, Brazil
University of Oeste de Santa Catarina (UNOESC)
Chapecó, Brazil
schreiner.geomar@posgrad.ufsc.br

Denio Duarte
Federal University of Fronteria Sul
Chapecó, Brazil
duarte@uffs.edu.br

Guilherme Dal Bianco
Federal University of Fronteria Sul
Chapecó, Brazil
guilherme.dalbianco@uffs.edu.br

Ronaldo dos Santos Mello
PPGCC - Federal University of Santa Catarina
Florianópolis, Brazil
r.mello@ufsc.br

ABSTRACT

Several application domains deal with the management of massive data volumes and thousands of OLTP transactions per second. Traditional relational databases cannot cope with these requirements. NewSQL is a new generation of databases that provides both high scalability and availability and ACID properties support. Besides, it is a promising solution to handle these application data management needs. Although data partitioning is an essential feature for tuning relational databases, stills an open issue for NewSQL systems. In this paper, we propose a hybrid partitioning approach for NewSQL databases that allows the user to define the vertical and horizontal data partitions. In order to determine what site will store each data fragment, we propose a hash function that considers schema information and data access statistics. Our experimental evaluation compares our hybrid VoltDB version against the standard VoltDB. The results highlight that our strategy increases the number of single-site transactions from 37% to 76%.

CCS CONCEPTS

• **Information systems → Relational database model**; *Data layout*; Distributed database transactions.

KEYWORDS

NewSQL, data partitioning, hybrid partitioning, VoltDB

ACM Reference Format:
Geomar A. Schreiner, Denio Duarte, Guilherme Dal Bianco, and Ronaldo dos Santos Mello. 2019. A Hybrid Partitioning Strategy for NewSQL Databases: The VoltDB Case. In *The 21st International Conference on Information Integration and Web-based Applications & Services (iiWAS2019), December 2–4, 2019, Munich, Germany.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3366030.3366062

1 INTRODUCTION

Application systems historically rely on OLTP transactions to perform small operations over the network, such as buying an item in an online store [23]. The number of users that use online stores has increasingly grown, and so the number of OLTP transactions. With the popularity of the *Web*, OLTP requests have changed as well to deal with online transactions, *e.g.* transactions over online games, social networks, or even large financial companies. These application domains are characterized by having a large number of user interactions with the system, generating a massive amount of data and multiple OLTP transactions per second.

For decades, traditional relational databases (RDBs) have been used efficiently to store and handle application data. However, they are not able to deal with a massive data volume and, at the same time, provide high availability and ACID guarantee. This is sometimes called the *Big Data challenge* [12, 23]. New database architectures are emerging to tackle the *Big Data challenge*, like *NoSQL DBs*, which offer high availability and scalability, and usually run as a service in the cloud. Different from traditional RDBs, NoSQL DBs are based on horizontal scaling, *i.e.*, new machines can be added to a cluster to increase the system performance.

NoSQL solves part of the problems related to Big Data management as they maximize availability rather than ACID support, using BASE approach (Basically Available, Soft state, Eventually consistency) [19]. Nevertheless, companies keep using RDBs for most of their applications because it is preferable to deal with the overhead of ACID assurance instead of being highly available [16]. *NewSQL* architecture has arisen to mitigate this situation by offering availability, scalability, and ACID support. *NewSQL* combines the best of both worlds: scalability and availability of NoSQL DBs with the ACID guarantee of RDBs to efficiently manage relational data [1, 16, 23]. Since NewSQL DBs are distributed in-memory DBs, they can handle thousands of *OLTP* transactions per second. Moreover, NewSQL DBs maintain partitions of the data on several nodes (or sites in NewSQL jargon) [6, 9, 11, 16, 25]. Nevertheless, they still suffer when multi-site (distributed) transactions are required since transactions are usually serialized and, further, executed, when it is possible, as single-sited. 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 the performance degradation for the multi-site transaction. 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) [2]. Horizontal partitioning divides a table into groups of tuples (*i.e*, selection operation). On the other hand, vertical partitioning divides a table into groups of disjoint columns (*i.e*,projection operation), and each group becomes a partition. Vertical partitioning improves the performance of queries that project only columns of a given partition, that is, it is suitable for OLAP applications. Instead, horizontal partitioning is more explored in OLTP solutions, and all the commercial NewSQL systems apply this strategy [2, 13]. Some academic NewSQL systems explore vertical partitioning in OLTP transactions [3], as well as hybrid partitioning for OLAP transactions [4]. For example, VoltDB provides ACID properties over a distributed architecture designed for high availability and scalability. Its architecture is an in-memory, shared-nothing and distributed RDB. The VoltDB[1] allows only the user to partition their data horizontally through the SQL DDL language. Regarding the NewSQL initiatives that explore the vertical partition for NewSQL systems, to the best of our knowledge, the use of hybrid partitioning in OLTP applications remains an open issue for NewSQL DBs.

In this paper, we introduce *Hybrid VoltDB*, a novel hybrid partitioning strategy for NewSQL systems developed on top of the VoltDB architecture. VoltDB was chosen because it is one of the most prominent NewSQL systems, and there is an open-source community edition [17]. Our approach extends VoltDB to work with vertical and both horizontal and vertical (hybrid) partitioning. We take advantage of the original *range hash* function, usually implemented for horizontal partitioning, to implement our vertical and consequently hybrid partitioning approaches. This function is responsible to route the correct fragment of data to each available node/site. To provide the hybrid partitioning we: *(i)* extend its DDL to allow the definition of vertical partitioning; *(ii)* modify the *hash function* to consider schema information and access statistics to split a table in vertical fragments to be stored in different sites; and *(iii)* change the execution of the SELECT statement to allowing the processing the vertical table fragments and, when necessary, to join these fragments. The extension of VoltDB for accepting vertical partitioning, as well as an experimental evaluation, are the main contributions of this work.

The rest of this paper is organized as follows. Section 2 gives a background about data partitioning, NewSQL DBs and the VoltDB. Section 3 is dedicated to the related work. Section 4 details our approach and Section 5 presents the experimental evaluation. Finally, Section 6 concludes the paper.

2 BACKGROUND

This section gives a brief background about data partitioning, as well as NewSQL DBs and the VoltDB.

[1] version 9.0

2.1 Data Partitioning

In a distributed environment data partitioning defines the data sharing, *i.e.*, how data are organized and allocated for each node on a distributed system. Two options are usually adopted to store data: *(i)* to keep the whole table in one site, or *(ii)* to split the table into fragments, maintaining each one in a different site. In the first option, the problem is that the table is not a suitable unit [13]. If each table is stored in different sites, a high number of additional remote data access are required to process join operations. Additionally, the maintenance of ACID properties becomes more complicated. Moreover, replicate all the tables in every site generates a high volume of replicated data, which can lead 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 [13].

Each fragment is considered as a partition of the data [5]. Usually, partitioning techniques can be summarized into three categories: *horizontal*, *vertical* and *hybrid* [22]. *Horizontal* partitioning consists of splitting a relation into other sub-relations (a subset) [22]. A horizontal fragment of a relation R consists of all R tuples that satisfy a partitioning function f_i [13]. This function f_i can be developed as an automated process that analyzes the DB schema and statistics about data accessing to infer properties (aka $minterms$) and groups 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 *director*. Blue lines belong to the partition P_1 ($f_i(director{=}2)$) and yellow lines to the partition P_2 ($f_i(director{=}1)$).

There are two main ways to split data in horizontal partitioning: *range* or *hash* [13]. *Range* partitioning defines categories and stores them on different nodes [10, 22]. A specific node is responsible for assigning the tuples among other nodes and maintaining mappings to locate data. This technique can cause a load unbalance between nodes because some categories may contain more tuples than others. *Hash* partitioning usually assumes a ring organization of the system nodes, where each node is responsible for a keys range. It also balances the nodes load through the analysis of statistics that complement the hash function. 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.

A relation R can also be vertically partitioned by considering projections $R_1, R_2, \ldots, R_n \subseteq R$ over different sets of attributes [13, 15]. The purpose is to group attributes into fragments that are often accessed together by applications [22]. However, this partitioning approach is complex since the choice of the fragments influences the performance of the system. 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.

In a real-world application, a horizontal or vertical partitioning of a database schema sometimes is not sufficient to satisfy all of its data manipulation requirements [13]. So, a *hybrid* partitioning is needed. In this case, horizontal and vertical partitioning are simultaneously applied over a table. Each fragment maintains a set of rows with a specific set of columns, which is stored in a node and are useful for queries that intend to retrieve such a table subset.

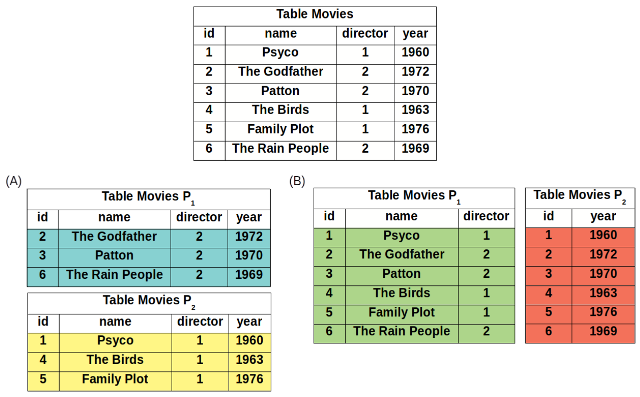

Table Movies

id	name	director	year
1	Psyco	1	1960
2	The Godfather	2	1972
3	Patton	2	1970
4	The Birds	1	1963
5	Family Plot	1	1976
6	The Rain People	2	1969

(A)

Table Movies P_1

id	name	director	year
2	The Godfather	2	1972
3	Patton	2	1970
6	The Rain People	2	1969

Table Movies P_2

id	name	director	year
1	Psyco	1	1960
4	The Birds	1	1963
5	Family Plot	1	1976

(B)

Table Movies P_1

id	name	director
1	Psyco	1
2	The Godfather	2
3	Patton	2
4	The Birds	1
5	Family Plot	1
6	The Rain People	2

Table Movies P_2

id	year
1	1960
2	1972
3	1970
4	1963
5	1976
6	1969

Figure 1: Data partitioning: (A) horizontal partitioning; (B) Vertical partitioning.

2.2 NewSQL Databases and the VoltDB

NewSQL DBs are RDBs that supply the demand for large OLTP workloads without renouncing the ACID properties [12, 16, 23]. As presented before, this new DB generation transcends the traditional RDBs by incorporating important features to Big Data management, such as high availability and scalability [10].

VoltDB is a *built-from-scratch* in-memory NewSQL DB based on *H-Store* [11] (the first NewSQL system) architecture. It has been developed to support large volumes of data and provide a full ACID guarantee for transactions and data streaming. VoltDB architecture consists of a *cluster* where the data is kept exclusively in main memory. This *cluster* is composed of two or more participating hosts in the same domain. A host, in turn, is a physical machine that stores one or more *sites*, and a site is the smallest portion of the architecture. A site is, basically, a *daemon* that receives OLTP requests and stores data in one partition. Each host has more than one processing core, and each *site* runs in one of these cores without sharing resources with other *sites*.

Figure 2 shows the architectural diagram of VoltDB. We see several sites that may be executing simultaneously. Each *site* is composed of three modules: (*i*) the *Single Partition Scheduler* is responsible for scheduling the execution of operations over the data partition, (*ii*) the *Stored Procedure Runtime* that stores all procedures or transactions that can be executed over the data, and (*iii*) the *Execution Engine* that properly execute the operations over the data. Each host of the cluster (*e.g.*, *Host 1*, *Host 2*, and *Host 3* in the figure) runs more than one site simultaneously. Figure 2 shows four sites per host, each one storing a data partition. The communication between sites, even between sites located on the same host, is performed by the *Message Fabric*.

The *Client Interface* module receives all application or user commands sent to the cluster. Each command is processed by *SQL/DDL Compiler & Query Optimizer*, which generates an optimized query plan. The *Schema & Configuration Manager* module stores schema information, that is, the dictionary, and the localization of each data share (a site). The generated query plan is decomposed into a set of operations. Each operation is sent to the site that stores the data target of the operation.

The *Multi-Partition Initiator* (MPI) module executes the initialization of each partition and the management of the transactions that access data in different partitions. The current version of VoltDB supports only horizontal partitioning by using a hash strategy. The function is based on a consistent *hash algorithm* and *data statistics* to split the table and keep the system workload balanced. The algorithm executes a hash function over the partition key (the partition column value) using the total number of partitions in the system. The result is considered as an option to store the tuple. The algorithm also analyzes the statistics and adequate the workload balance to decide on which site to store the tuple. Users can partition tables at creation time. After data insertion, the table partitioning schema is immutable. If a modification is needed, the table must be recreated, and the data migrated to this new table.

As previously stated, data is stored in a shared-nothing architecture, and each site stores a share of the data. Each site receives transactions forwarded by the *Schema & Configuration Manager*. This transaction is serialized and executed in a single site that ensures the ACID properties. If a transaction t_i accesses data in multiple sites, one of them is elected as coordinator. This coordinator manages t_i execution, ensuring the ACID properties.

VoltDB supports a *lock-free* concurrency control protocol. As it relies primarily on memory for data storage, each site stores a share of data and most of the transactions run in a single site. Thus, VoltDB does not need to implement complex concurrency control mechanisms. Only when VoltDB performs a distributed transaction, the MPI defines locks to coordinate the process [24].

Additionally, VoltDB provides high availability as well as fault tolerance using consistent replication of each data partition. Each data partition is replicated over the cluster, so if one *site* fails, another site that stores a partition replica takes its place. In Figure 2, *Execution Site 1* stores data partition *A* and *Execution Site 5* stores a replica of partition *A* (*Partition A′*). The replication model of VoltDB allows the user to define a *k-factor* (K-Safety) that controls the number of replicas in the system. The *k-factor* represents the k number of *sites* that the system allows failing without any data loss. Each replica is an identical copy of a data partition. The MPI sends operations for each site and their respective replicas, and the processes are executed synchronously in the same order on each site. In the figure, the k-factor is set to 1 since there is one replica for every partition, that is, if one host is down (*e.g.*, *Host 2*), another host containing the replicas (*e.g.*, *Host 3*) will answer the requests.

3 RELATED WORK

There are some works in the literature that propose partitioning solutions for NewSQL DBs, and each one focuses on different perspectives of the problem. *S-Store* [6, 14], *Horticulture* [17], *Schism* [7], *E-Store* [25], *Accordion* [20] and *Clay* [21] present automatic data partitioning strategies based on the current system workload. *Horticulture* [17] and *Schism* [7] consider stored procedures and schema extract rules to help user to properly partition their data. *E-Store* [25], *Accordion* [20] and *Clay* [21] propose tools that analyze the system workload and define rules for improving data partitioning to reduce the amount of distributed transactions; however, they deal only with horizontal partitioning.

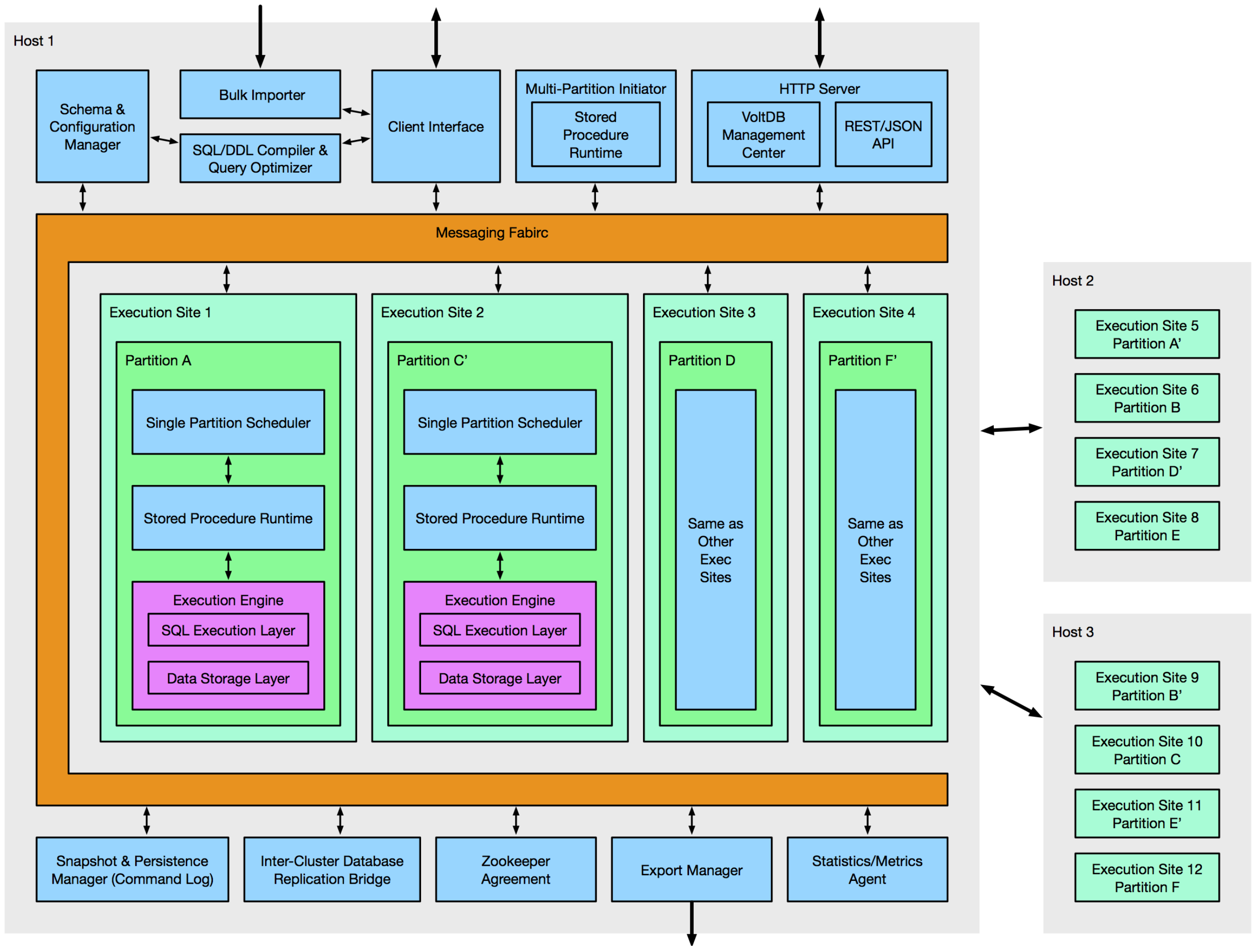

Figure 2: *VoltDB* architecture.

Other approaches like *RubatoDB* [26], *Teradata* [2], *Rabl* [18], and *Archipelago* [4] consider horizontal and vertical partitioning for system performance improvement. However, they were not developed for ordinary NewSQL DBs (OLTP-based systems). Instead, they are suitable for applications that execute both OLTP and OLAP transactions. Data requested by OLTP transactions are stored in a horizontal partition scheme. On the other hand, the vertical partitioning is proposed for data related to OLAP transactions, *i.e.*, these approaches use vertical partitioning exclusively to reduce the amount of main memory used to handle OLAP transactions.

Different from related work, we explore hybrid partitioning as an option to decrease the number of distributed transactions when accessing data. Our main idea is to modify the typical NewSQL horizontal hash partitioning scheme also to consider a hybrid (vertical and horizontal) partitioning scheme and eliminate the need for maintaining information about the location of fragments.

4 DATA PARTITIONING APPROACH

This section details our hybrid data partitioning approach for NewSQL DBs. We first give a formal overview of it and, in the following, we describe its development in the VoltDB architecture.

4.1 Approach Overview

In a NewSQL system, data partitioning usually applies horizontal *hash function* to split data. In this work, we extend this *hash function* to also consider vertical partitioning, *i.e.*, to route data over the cluster considering a hybrid partition scheme. Our primary purpose of using hybrid data partitioning is to allow a domain expert user to properly distribute their data to minimize the number of distributed transactions in query operations.

Our approach allows users to use SQL DDL clauses to define data partitions. During a tuple insert operation, the NewSQL system sends the operation to every site on the cluster. Each site receives it and verifies the possibility to store the tuple. The site executes this verification using a *hash function* that returns the *id* of the site that is responsible for storing the tuple. Our approach extends this function to employ vertical and hybrid data partitioning. This extended function uses schema information and statistics about the system workload to allocate data that are accessed together (*e.g.*, by join operations) in the same site.

For the sake of understanding, this section first defines some preliminary definitions that our approach exploits. In the following, we define the algorithms that execute the data distribution process.

Definition 4.1. **(Relation)**. A database relation $r = (T_r, A_r)$ is a set of tuples $T_r = \{t_1, \ldots, t_n\}$ and a set of attributes $A_r = \{a_1, \ldots, a_m\}$.

Definition 4.2. **(Fragment)**. A fragment $f = (T_f, A_f)$ is a set of tuples $T_f = \{t_1, \ldots, t_p\}$ and a set of attributes $A_f = \{a_1, \ldots, a_q\}$ of a database relation r, where $(f_i.T_f \cup f_i.A_f) \subset (r.T_r \cup r.A_r)$.

Definition 4.3. **(Partitioned Database)**. A partitioned database holds a set of relations R, where a relation $r \in R$ can be split into a set of fragments $F = \{f_1, f_2, \ldots, f_k\}$, and each fragment $f_i \in F$ is usually stored in a partition of the system where k is the number of partitions of the database, so that a horizontal partitioned database has $f_i.T_f \subset r.T_r$ and $f_i.A_f = r.A_r$, a vertical partitioned database has $f_i.T_f = r.T_r$ and $f_i.A_f \subset r.A_r$, and a hybrid partitioned database has $(f_i.T_f \cup f_i.A_f) \subset (r.T_r \cup r.A_r)$.

According to Definition 4.2, a fragment f is a subset of a database relation r (Definition 4.1) so that f maintains less tuples or less attributes than r. Definition 4.3, based on Definition 4.2, states that a partitioned database can split relations in several fragments based on the number of allowed partitions.

Definition 4.4. **(PHash)**. *PHash* is a partitioning function that applies a hash function to return the sites where the fragments of a new relational tuple must be stored with the signature $PHash(t_y, \mathcal{R}, \mathcal{P})$, where t_y is a tuple of a relation $\mathcal{R}$ that must be stored in the database, $\mathcal{P}$ is the set of attributes $A_x \subseteq \mathcal{R}.A_r$ that defines the partition key (for horizontal partitioning).

As shown in the Definition 4.4, *PHash* returns the site *id* where the new tuple t_y must be stored. To accomplish that it uses information about the schema and where the vertical partitions are located. This information is available in the data catalog for all the sites – Algorithm 1 sketches *PHash* function.

Given a tuple t_y to be inserted (insert operation) into a relation $\mathcal{R}$, t_y is sent to all the sites of the cluster (*i.e.*, a broadcast). Every site runs *PHash* over the partition key to check whether or not it stores a fragment of t_y. *PHash* returns the *id* of the site that must store the fragments of t_y. As stated previously, we change the standard NewSQL *PHash* function to also consider vertical partitioning. When *PHash* is executed, it first accesses the database metadata in *s* to check the partitioning type (see Algorithm 1). If the type is hybrid or vertical, it breaks t_y in several fragments, where each fragment f_i represents a subset of t_y that must be stored into a given site. Notice that every site executes *PHash*, only those that *id* is the same returned by *PHash* store the fragment, the others ignore the operation. This approach is inherited from VoltDB. In the following, we present an extract of the code running when a tuple is requested to be inserted (*i.e.*, a deamon process running on every site).

```
$(r, t) \leftarrow (relation, tuple)$ // incoming relation and tuple
$p \leftarrow extractPartitionKey(t, r)$
if $SiteId = PHash(t, r, p)$ then
  $storeInFragment(t, r)$
end if
```

According to Algorithm 1, each site splits the tuple into fragments based on the number of candidates, which is the number of user-created vertical partitions for the target relation.

```
Result: Return the Id of the site that stores the part of the
        tuple
s = loadMetadata();
if r.isVerticallyPartitioned(s) then
    numberOfCandidates = r.numberOfVerticalPartitions(s);
    tupleFragments = split(t,numberOfCandidates);
else
    numberOfCandidates = 1;
    tupleFragment = extractFragment(t);
end
for i = 0, i < numberOfCandidates, i++ do
    verticalSiteIds =
      s.getVerticalSites(tupleFragments[i].params);
    Id = chooseHorizontalSite(tupleFragments[i],
      verticalSiteIds);
    if SiteId == Id then
        return Id;
    end
end
return -1;
```

Algorithm 1: PHash function.

The current version of our approach, using hybrid partitioning or pure vertical partitioning, allows one vertical partition and one horizontal partition for each table. This limitation is because VoltDB internal structures only support one horizontal partition per table, and change these internal structures is very complicated since the entire system storing structures relies on this principle. Additionally, each site tests itself as a candidate to verify whether or not a fragment of the tuple needs to be stored on it. The function *chooseHorizontalSite* accesses metadata (*e.g.*, foreign keys, statistics about co-accessed tables, and frequent joins) of the system to return the correct site *id* to store the fragment. This function runs the original hash algorithm for horizontal partitioning of VoltDB. Remark that vertical partitioning is classically implemented as a centralized model where a node routes the parts of the tuple for the storage sites, and holds all the information of each chunk stored and where they are stored. Our approach eliminates the need for a centralized node with complex information since each site is able to decide (based on *PHash*) what fragment of the tuple it must store.

4.2 Hybrid-VoltDB

The current version of VoltDB (9.0) supports only user-defined horizontal data partitioning. Its partitioning strategy is based on a hash function that stores data in different sites considering load balancing and functional restrictions (foreign keys) requirements. On taking advantage of the VoltDB architecture and horizontal hash function, we change its standard hash function with our approach (as detailed in the previous section) to allow the definition of vertical and hybrid partitioning. In this section, we present how vertical and hybrid partitionings are implemented.

First of all, we change the VoltDB SQL parser to add tokens for vertical partitioning. Figure 3(A) shows the original table partition DDL clause schema for VoltDB, and Figure 3(B) shows our new partition statement. The new clause allows the definition of the columns that will be allocated in the same partition. It is worth mentioning that we do not define a new statement for hybrid partitioning. Instead, the user is free to define horizontal, vertical, or both partition for the same table.

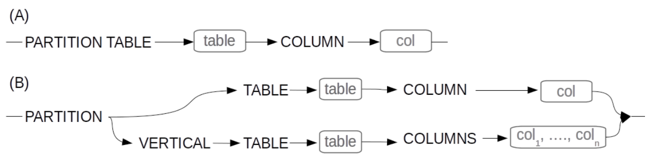

Figure 3: SQL DDL partitioning statement.

Transactions in VoltDB can be executed in two ways: *local* or *distributed.* A *local* transaction executes operations on a single site. Otherwise, it is a *distributed* transaction. A tuple *insert* operation is executed locally when the relation is horizontally partitioned, and the database has no replicas. If there are replicas, this operation is considered a distributed transaction. Independently if the transaction is local or distributed, all the sites receive the insert statement, and based in the hash function only the sites that store the correct partition (and the replicas) execute the operation. We also changed VoltDB to consider inserts in vertically partitioned tables as well as a distributed transaction.

When an insert statement for a new tuple t_y is sent to VoltDB, we verify if the target relation $\mathcal{R}$ is vertically partitioned. If so, the statement is marked as a distributed transaction. An execution plan with the insert details (columns, values, fragments) is generated and sent for all the cluster. Each site in the cluster receives the execution plan and executes the new $PHash$ function (Algorithm 1). Only the sites that hold parts of $\mathcal{R}$ will store t_y fragments. Besides, the replication of hybrid or vertical partitioned relations is automatically performed by VoltDB. Thus, all sites that hold $\mathcal{R}$ replicas also store t_y, as stated in Section 2.2.

Besides changing the way that VoltDB stores data, we also improve the methods that retrieve data. In the original VoltDB, each fragment represents a full tuple. So, when a coordinator executes a *select* statement, all sites receive the execution plan, and each site returns to the coordinator the fragments that satisfy the query condition. The coordinator is responsible for unifying all fragments and returns the result to the user. In *Hybrid-VoltDB*, a tuple can be split in multiple fragments depending on the vertical partitioning scheme. So, when executing a *select* statement, the coordinator also sends the execution plan to all sites. However, the coordinator has to additionally join the received fragments that represent parts of the resulting tuples. In the current version of *Hybrid-VoltDB*, the fragments are joined by the primary keys.

5 EXPERIMENTS

This section presents a set of experiments conducted on VoltDB and Hybrid-VoltDB in order to demonstrate the effectiveness of our hybrid approach. We consider the *OLTP-Bench* [8] in our experiments. It is a benchmark suite that allows users to execute benchmarks over different commercial database systems. It generates a transaction queue to execute according to the chosen benchmark specification and a user-supplied configuration file. This queue is executed in parallel by many *workers* (emulate users) configured in the input file. During the execution of the tests, *OLTP-Bench* collects and returns the execution statistics. We compare our approach with different partitioning schemes against the standard *VoltDB* partitioned horizontally. This section first presents the methodology of the experiments and then their results and analysis.

5.1 Experiment Settings and Methodology

The experiments were performed on a server machine with four Intel(R) Xeon(R) CPU E7- 4850 (2.00GHz) processors with 128 GB RAM, running Linux 4.15.0-50 kernel (Ubuntu 18.04.2 LTS distribution). In the server, we use Docker[2] to emulate three different machines (hosts) connected by a virtual network. Every machine runs its own VoltDB instance, and the three are configured to be a VoltDB cluster. The cluster is configured without replication factor (*k-factor* = 0) and is set to use 8 cores sites for each container. In the experiment, we use OLTP-Bench to run the TPC-C benchmark[3]. The OLTP-Bench ran in an external machine (Intel Core i5-2430M processor with 8 GB DDR3 1066mHz RAM, 240GB Scandisk SSD) connected to the cluster by a LAN connection (100 MB).

The TPC-C emulates a set of terminal operators executing a series of operations. TPC-C focuses on the main activities (transactions) of an *order-entry* application. This application defines transactions for entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. The transactions are executed over nine types of tables with a wide range of tuples and sizes.

We executed our experiments in two parts. The first one focused on the system performance, evaluating the throughput of the original VoltDB and comparing it with our hybrid version. In the second part, we focus on the number of distributed transactions. We evaluate whether or not the hybrid approach reduces the number of distributed transactions.

We executed *OLTPBench* with the same TPC-C configuration for each experiment. We set 2 as the scale factor, 20 terminals running simultaneously, a timeout of 60 seconds, and a transaction rate of 10, 000. We ran the *OLTPBench* ten times with different partitioning schemes:

- Standard VoltDB without data partitioning, called *No Partition (i).*
- Standard VoltDB with the partitioned tables *STOCK*, *ITEM* and *ORDER*, each one partitioned by the column *W_ID* that represents the warehouse, which is part of the primary keys of all the tables and it is present in all the queries, called *Hor Part(ii).*

[2]http://www.docker.com/
[3]http://www.tpc.org/tpcc/

- Hybrid-VoltDB with vertical partitioning of the tables *STOCK*, *ITEM* and *ORDER*, called *Vert Part (iii)*.
- Hybrid-VoltDB with vertical partitioning of the tables *COSTUMER*, and *DISTRICT*, called *Vert Part (iv)*.
- Hybrid-VoltDB applying a hybrid partition scheme: a horizontal partitioning of the table *ORDER_LINE* by the column *OL_W_ID*, and a vertical partitioning of the tables *ORDER_LINE* and *STOCK*, called *Hybrid Part (v)*.

5.2 NewSQL Database System Performance

This section presents the results with relation to the number of transactions executed per second. They are shown in Figure 4 for all the five partitioning schemes.

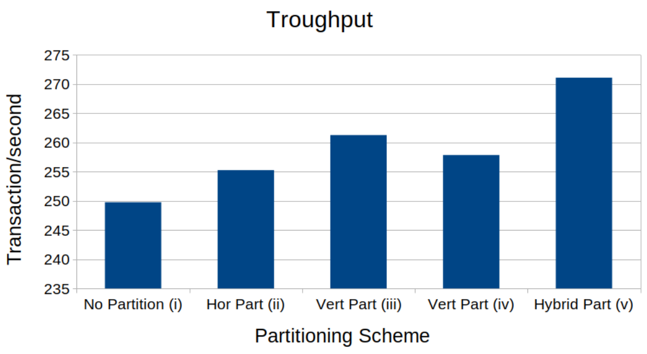

Figure 4: Transactions per second for the partitioning schemes.

As Figure 4 shows, Scheme *No Partition(i)* obtained the worst result by handling 249.75 transactions per second. When VoltDB works without data partitioning, all sites stores all the data (i.e., each site have a complete version of the data). Due to it, INSERT, UPDATE and DELETE operations become more expansive considering that all the sites need to execute the same operation in order to guarantee the ACID properties. The standard VoltDB horizontal partitioning (scheme *Hor Part(ii)*) achieves a better result than the previous one, but it was slower than the other schemes. It executed 255.25 transaction per second. Once data are distributed over the cluster, update operations are executed as single-sited.

The results for the other schemes using our vertical or hybrid data partitioning outperformed the VoltDB horizontal partitioning and the no partitioned scheme. The hybrid partitioning scheme is the most prominent of them. As stated before, we consider the most frequent transactions in the DB system to define the data partitions. Even without all tables partitioned, our hybrid approach results were 8.5% faster than the no partitioned system. Also, notice that since the cluster (three docker containers) are running in the same machine, the communication between the hosts is way faster than a regular cluster connected by internet connection. We believe that this small difference would be more significant if the cluster were placed in physically different machines because, in this scenario, the latency is very low.

5.3 Distributed Transactions

This section shows the number of distributed transactions on each partitioning scheme. We consider, as a distributed transaction, a transaction that touches data in more than one site and needs a coordinator that waits until all sites execute an operation, and eventually merges the results. For this experiment, the *no partition* scheme was ignored, giving that all data is replicated and all operations are executed in single site. INSERT and DELETE operations were not also taken into account because TPC-C execution has a loading phase that executes inserts operation massively, and the framework does not consider this phase to compute the results. Thus, we evaluated only SELECT and UPDATE operations that could be executed as single sited transactions, *i.e.*, operations executed only in one site. The bar graph of Figure 5 shows the percentage of the single sited transactions for each scheme with relation to all transactions handled by the scheme.

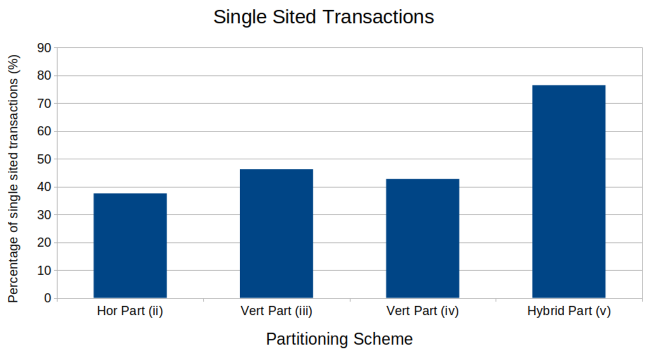

Figure 5: Percentage of single-sited transactions.

The results highlight that the hybrid scheme (*Hybrid Part (v)*) performs 76.45% of the transaction as single-sited. On the other hand, the horizontal partitioning scheme (*Hor Part(ii)*) and the two vertical partitioning schemes (*Vert Part(iii)* and *Vert Part(iv)*) obtain very similar results, around 40%. It means half of the transactions performed by them are distributed. Moreover, the horizontal partitioning scheme gets the worst result achieving only 37.5% of single sited transactions.

The intuition behind these results are as follows. In the horizontal partition scheme, each tuple is stored in only one fragment. This storage structure increases the complexity of the allocating process of fragments in a way that minimizes the number of distributed transactions. For example, given two tables T_1 and T_2 and two fragments of T_1: $ft1_1$ and $ft1_2$. Consider that two different transactions t and t', t is executing the operation $ft1_1 \bowtie \sigma_\theta(\pi_{(\alpha)}(T_2))$ and t' is executing the operation $ft1_2 \bowtie \sigma_\theta(\pi_{(\beta)}(T_2))$, where α and β are distinct projections (*i.e.*, $\alpha \neq \beta$). Even if we consider an optimal case, for example, T_2 and $ft1_1$ are stored in the same site, one of the transactions is executed distributed, since $ft1_2$ is stored in a different site.

An optimal case for hybrid partition could be: given two sites s_1 and s_2, s_1 stores $ft1_1$ and $\pi_{(\alpha)}(T_2)$, and s_2 stores $ft1_2$ and $\pi_{(\beta)}(T_2)$. In this case, both transactions would be single-sited, and the overall performance would be better, as our experiment showed.

The use of hybrid data partitioning allows a more fine-grained partitioning of the tables, exploring the advantages of both data partitions schemes (horizontal and vertical). Considering that the hybrid partition scheme creates fragments more specialized (created by the user with DDL statements), distributed transactions are less likely to happen.

These experiments revealed that our hybrid approach executed significant more single sited transactions than other partitions schemes, *i.e.*, it reduces the number of distributed transactions in a larger scale.

6 CONCLUSION

This paper introduces *Hybrid-VoltDB*, an approach for managing hybrid data partitioning materialized in $VoltDB$, a pioneer and the most popular NewSQL database system. Our contribution with *Hybrid-VoltDB* is to fill a gap for OLTP-based NewSQL databases, which offer only horizontal data partitioning. On including additional support to vertical partitioning, our hypothesis is to improve the performance of distributed OLTP transactions.

To provide a hybrid partitioning, we modified the hash-based partitioning function of VoltDB to accomplish a efficient vertical and horizontal fragmentation of a relational table based on the schema information and workload statistics. We also modified the partition statement of the VoltDB SQL DDL to allow the definition of a vertical partitioning for a table beside the default horizontal partitioning. All of these improvements are evaluated and validated through a set of experiments. The results demonstrate the efficiency of *Hybrid-VoltDB* in two flavors: in terms of the performance of distributed transactions and the overall transaction processing of the native VoltDB even without data partitioning. These results show that our approach is promising and confirm our hypothesis.

Future work includes the evaluation of our approach with other benchmarks and running tests over a cluster physically distributed. We also have in mind the development of an automated approach that considers workload to define new partitioning schemes to reduce the number of distributed transactions.

ACKNOWLEDGMENTS

This work was partially supported by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

REFERENCES

[1] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. 2009. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 922–933.

[2] Mohammed Al-Kateb, Paul Sinclair, Grace Au, and Carrie Ballinger. 2016. Hybrid Row-column Partitioning in Teradata&Reg;. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1353–1364.

[3] R. R. Amossen. 2010. Vertical partitioning of relational OLTP databases using integer programming. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. 93–98. https://doi.org/10.1109/ICDEW.2010.5452739

[4] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 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)*. ACM, New York, NY, USA, 16.

[5] E. Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Feb 2012), 23–29.

[6] Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, John Meehan, Andrew Pavlo, Michael Stonebraker, Erik Sutherland, Nesime Tatbul, Kristin Tufte, Hao Wang, and Stanley Zdonik. 2014. S-Store: A Streaming NewSQL System for Big Velocity Applications. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 4.

[7] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. 2010. Schism: A Workload-driven Approach to Database Replication and Partitioning. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 48–57. https://doi.org/10.14778/1920841.1920853

[8] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec. 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[9] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. 2015. Squall: Fine-Grained Live Reconfiguration for Partitioned Main Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 299–313.

[10] Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari, and Miriam AM Capretz. 2013. Data management in cloud environments: NoSQL and NewSQL data stores. *Journal of Cloud Computing: Advances, Systems and Applications* 2, 1 (2013), 22.

[11] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[12] Rakesh Kumar, Neha Gupta, Shilpi Charu, and Sunil Kumar Jangir. 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)*.

[13] Patrick Valduriez (auth.) M. Tamer Özsu. 2011. *Principles of Distributed Database Systems, Third Edition* (3 ed.). Springer-Verlag New York.

[14] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (Sept. 2015), 12.

[15] Shamkant Navathe, Stefano Ceri, Gio Wiederhold, and Jinglie Dou. 1984. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems (TODS)* 9, 4 (1984), 680–710.

[16] Andrew Pavlo and Matthew Aslett. 2016. What's Really New with NewSQL? *SIGMOD Rec.* 45, 2 (Sept. 2016), 45–55.

[17] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 61–72.

[18] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 315–330.

[19] Pramod J Sadalage and Martin Fowler. 2012. *NoSQL distilled: a brief guide to the emerging world of polyglot persistence.* Pearson Education.

[20] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. 2014. Accordion: Elastic Scalability for Database Systems Supporting Distributed Transactions. *Proc. VLDB Endow.* 7, 12 (Aug. 2014), 1035–1046.

[21] Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. 2016. Clay: Fine-grained Adaptive Partitioning for General Database Schemas. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 445–456.

[22] Gitanjali Sharma and Pankaj Deep Kaur. 2015. Architecting Solutions for Scalable Databases in Cloud. In *Proceedings of the Third International Symposium on Women in Computing and Informatics (WCI '15)*. ACM, New York, NY, USA, 469–476.

[23] Michael Stonebraker. 2012. New Opportunities for New SQL. *Commun. ACM* 55, 11 (Nov. 2012), 10–11. https://doi.org/10.1145/2366316.2366319

[24] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.

[25] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[26] Li-Yan Yuan, Lengdong Wu, Jia-Huai You, and Yan Chi. 2015. A Demonstration of Rubato DB: A Highly Scalable NewSQL Database System for OLTP and Big Data Applications. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 6.