TÉCNICO LISBOA

Database Partitioning via Symbolic Execution

Francisco Jorge Silva Santos Rola

Thesis to obtain the Master of Science Degree in

Information Systems and Software Engineering

Supervisor(s): Prof. Paolo Romano
Prof. Miguel Ângelo Marques de Matos

Examination Committee

Chairperson: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Supervisor: Prof. Miguel Ângelo Marques de Matos
Member of the Committee: Prof. Vasco Miguel Gomes Nunes Manquinho

January 2021

I would like to dedicate this thesis to a few special people. First, I dedicate it to my mother who always worked her absolute hardest to make sure I always had everything I needed. Without her guidance and hard work, I would be nowhere near where I am right now. I would also like to thank three people that took care of me since I was born during my mother's work hours. Thank you Titi, Titó and Kiki. I am very thankful for everything you did for me, everything you taught me. You always pushed me to be better, to aim higher, to not quit. Thank you I am also thankful to my stepfather. Without him, I don't know how the first year in university would turn out. Not only that but he has been my father since I can remember. I would like to dedicate this work to my girlfriend who is always there for me when I need it most, encouraging me to achieve my own goals, lifting my mood when everything goes wrong and making me feel special every single day.

Thank you.

Acknowledgments

I would like to thank the university for providing such a good place to not only learn my craft but also learn how to be the better version of myself. A lot of thinking goes into the structure of each course and the organization levels are always on point. I am very thankful to Professor Paolo Romano, Professor Miguel Matos, Dr Shady Issa and Dr Antti Hyvärinen for all the work they have put into this project. Long discussions and a lot of brainstorming, it was a pleasure to work alongside all of them. I would like to thank the reader for taking the time to read this thesis.

Resumo

As bases de dados são uma peça fundamental para qualquer sistema que necessite de guardar dados de forma persistente. Com a emergência de sistemas de larga escala, as bases de dados que os suportam necessitam de abordar todas as questões de escalabilidade inerentes. Em particular, não é possível replicar os dados em todas as réplicas que constituem um sistema, sendo necessário distribuir os dados entre as réplicas, surgindo assim o particionamento de dados Trabalhos de investigação anteriores nesta área requerem extratos de execução do sistema para adquirir informação referente aos dados processados de forma a conseguir particionar os dados. Este trabalho sugere utilizar execução simbólica com o intuito de alcançar um esquema de particionamento de dados ideal. Com este objetivo em vista, formulamos o problema como um problema de particionamento de um grafo que utiliza a informação obtida durante a análise simbólica. Depois de resolver o problema de particionamento de um grafo, aplicamos o esquema de particionamento resultante â base de dados por forma a obter partições ideias de acordo com a nossa formulação do problema. Os resultados obtidos para o benchmark TPC-C são bastante satisfatórios tendo em consta que estão de acordo com a melhor solução conhecida para este benchmark.

Palavras-chave: bases de dados, particionamento de dados, execução simbólica, particionamento de grafos

Abstract

Databases are a fundamental piece of the jigsaw behind any system that requires persistent storage. With the emergence of large scale database systems, it is necessary to address all the scalability issues that come with them. In particular, it is not possible to fully replicate every data item across every single replica in a system hence the need for data partitioning. Previous work in this area relies on execution traces and prior knowledge of the data being processed in order to perform data partitioning. In this work, we propose to leverage Symbolic Execution to replace these techniques and explore symbolic formulas to reach an optimal data partitioning scheme. In order to achieve this, we formulate a graph partitioning problem that leverages all the information obtained during symbolic analysis. After obtaining the partitioning scheme from a graph partitioner, we apply it to the database in order to achieve optimized partitions according to our formulation. The experimental results obtained for the TPC-C benchmark are very satisfying as they are aligned with what is known to be the best partitioning scheme.

Keywords: database, data partitioning, symbolic execution, graph partitioning

Contents

List of Tables

List of Figures

Nomenclature

Greek symbols

ϕ	Expression that states the range of input variables to the ρ expressions
ρ	Expression that evaluates to a data item identifier.
σ	Expression that describes a set of ρ formulas and corresponding ϕ

Chapter 1

Introduction

1.1 Motivation

In the modern days, due to the appearance of geo-replicated systems, it is necessary to place data across data centers in such a way that it can be accessed in a timely manner by all customers. These customers can be spread across the globe, however, they will always have a data center closer to them. A good system would distribute the data such that all clients have their data on a nearby data center. State of the art databases face many challenges in order to provide the user with a seemingly local store of nearly unbound dimensions. The flexibility constraints in said systems require them to adapt to unsteady demands which depend on many external factors such as user demand. The frequency a given item is accessed within a database is not easily predictable. Furthermore, the amount of data held by these stores raises scalability concerns. It is not feasible to have every single data entry fully synchronized across the various data centers spread throughout the globe. Consequently, it is necessary to provide the users with a system that has access times similar to what they have grown accustomed to when using local storage while guaranteeing disaster recovery, fault tolerance and scalability. As previously stated, full replication is not a feasible solution in this scenario, it is a simple approach but in the presence of an overwhelming amount of data, it simply does not scale.

A typical solution to solve this scalability problem is the so-called data partitioning, also known in the literature as data sharding. This technique involves developing a partitioning scheme that determines where is every data item placed and where is it replicated. By employing this strategy, it is possible to provide a fault-tolerant system which scales to the level required by modern day's databases. However, developing an optimal partitioning scheme is a non-trivial task due to the wobbling nature of clients' requests. It is necessary to know the data access patterns for each item in the database in order to group together items accessed by a given

transaction.

1.2 Topic Overview

There are many different strategies to solve the aforementioned problem. Reaching an optimal partitioning scheme will always require some sort of knowledge on the data being handled. One possible approach is to define the number of partitions one desires to have and then deploy a partitioning algorithm with an execution trace that comprises a representative workload of the queries requested by clients. Schism [CJZM10] applies this approach by leveraging a graph partitioner. While this technique provides good results, it requires an execution trace that is not always available in real systems and, additionally, may miss rare access patterns. An alternative approach is the usage of an oracle component as employed in Dynastar [LFEK+19]. This component is responsible for keeping a global view of the system and thus being able to perform optimal partition decisions. In order to achieve its goal, this system builds a workload graph by also leveraging a graph partitioner. It does not receive a representative workload as input, instead, it performs this optimization on the fly.

In this work, we address the data partitioning problem from a different point of view. Whereas previous systems require representative execution traces [CJZM10] or building complex graphs[LFEK+19] in order to obtain good data partition schemes, we choose to leverage Symbolic Execution (SE) to solve the problem. By exploring the fine-grained knowledge provided by a symbolic execution engine, we determine which records each transaction accesses and therefore have extra knowledge on data access patterns. Furthermore, the symbolic formulas obtained can be leveraged to formulate the partitioning problem as a graph partitioning problem, just like in the aforementioned systems.

1.3 Objectives

The goal of this project is to improve on current database partitioning approaches by reaching a data placement scheme capable of minimizing cross partition transactions and, therefore, maximize the system transactions throughput. In order to achieve this, we will leverage symbolic execution in a way it has not been used before, by combining symbolic analysis with a graph partitioner.

1.4 Thesis Outline

The rest of this document is organized as follows. In Section 2 we present the related work in the various research areas comprised in this thesis. Section 3 presents our proposed solution. In section 4 we present the result of our system evaluation. Section 5 concludes this thesis.

Chapter 2

Background

There are two different areas explored in this work. Namely, symbolic execution and data partitioning. The focus of this work is on data partitioning. In this section, we will give the reader an understanding of all techniques leveraged in recent works as well as provide background on the topics under study.

2.1 Symbolic execution

Symbolic execution is a program static analysis technique. It is capable of analyzing a program and determining what inputs trigger each program path to execute. Instead of relying on concrete inputs like normal programs, symbolic execution leverages an interpreter which follows the program execution utilizing symbolic values as inputs. As a result, symbolic analysis is capable of deriving expressions defined based on those symbolic inputs for expressions and variables in a program. It is also capable of stating the outcome of conditional branches in the program based on those symbols. Due to its nature, it is has seen increased use cases in software testing.

In order to give the reader a better understanding of the concepts involved, we provide an example of symbolic analysis in section 3 where we detail our system. However, in this section, we will present two different works. The first one, Prognosticator[Vie18], was developed using the same symbolic analysis tool as our work although with a completely different final outcome in mind. The second one[BCE08] discusses a known limitation with symbolic execution known as path explosion. This issue is also tacked by our solution hence the need to provide the reader with some background on the topic.

2.1.1 Prognosticator

Prognosticator proposes a solution that seeks to improve database throughput by leveraging symbolic execution.

In short, Prognosticator attempts to maximize transaction throughput by exploring parallelism while keeping the same serial order across replicas. In order to achieve this, it comprises two steps. The first phase is executed offline, during this stage Prognosticator exploits symbolic execution to build a base of knowledge on the transactions the system can be queried with, considering every possible input. This stage has as output a complete read and write set for that transaction, i.e. every data item that transaction accesses and the mode it accesses it in. In the second stage, the transaction profile generated earlier is leveraged at run time to perform transaction scheduling. By following this approach this system seeks to maximize the parallelism between non-conflicting transactions. Note that identifying if a transaction conflicts with another is as trivial as verifying if their read and write sets are disjoint or not.

2.1.2 RWSet

Path explosion is a known problem to exist with symbolic execution engines. When a program contains conditional branches and loops, or simply is too long, the amount of paths generated by a standard symbolic analysis is overwhelming and at times will take an unreasonable amount of time to execute. This problem is relevant in the context of our work since we intend to leverage symbolic execution and require it to finish in order for our system to proceed.

RWSet introduces two main ideas to tackle this problem however the main concept is very simple: prune redundant paths by tracking the memory locations read and written. This is achieved by not exploring paths that are known to not generate any behaviors despite how long they could be potentially explored for.

The first idea they introduce is very simple. If a given execution of a program, on a given point, comprises the same state as some previously seen execution, at that same program point, then it is safe to discard the current execution as it is guaranteed that it will not generate any new knowledge. This technique is sound and does not affect the output of symbolic execution.

Despite the first idea being clever by itself, it will not improve execution times in many scenarios as it is unlikely that two different executions reach the same point with the same state. This is where the second idea in this work comes in. If two programs only differ in program values that will not be subsequently read then those programs will produce the same subsequent effects and therefore can be considered equivalent in the context of RWSet.

In short, their implementation involves leveraging a constraint-based execution tool. By

creating a constraint cache they are able to truncate paths with the same constraints as some previous path. They all introduce the concept of live variables, this denotes the set of variables which are read by a program after a program point given a set of path constraints. Lastly, they refine the system by considering two constraint sets equal if all constraints comprising live variables are equal.

With the background knowledge provided by this section, it is now possible to introduce the data partitioning problem and understand how it relates to the systems previously presented.

2.2 Data Partitioning

Data partitioning is a technique that allows for the distribution of data across a set of machines. The data present in each machine is determined by the partitioning scheme computed by a data partitioning system. The main goal when performing data partitioning is to maximize the locality of database transactions, i.e. ensure that all the data required by a transaction is present within a single machine. Transactions that can not fully execute locally are known as distributed transactions, as they require data distributed across multiple machines. These transactions are also known as cross partition transactions in the literature. We will now present both of these systems.

Any system that seeks to perform data partitioning has a few aspects in common. First, the system needs some sort of input about the workload. This information can be given in the form of an execution trace or can be collected via online analysis of the system at run time. Second, these systems have a set goal, such as data balancing or request balancing. Third, the partitioning strategy can be based on hashing, in which case the data gets hashed to a partition via a hash function. The data can also be placed via a partition function that maps a given tuple to a machine. It is also possible to apply techniques such as machine learning to group certain data tuples together. This technique is known as range-predicate partitioning. Lastly, the output of the system is a data placement scheme which states the mapping between data items and partitions.

This section explores recent work in the data partitioning field, also known as data sharding. Schism[CJZM10] is the first system presented in this section. It seeks to produce an optimal partition scheme by leveraging an execution trace and graph partitioning techniques, this is an offline solution. Secondly, Dynastar [LFEK+19] is a recent work that also leverages a graph partitioner but builds it on the fly instead and thus works online. Lastly, Autoplacer [PRRR13] focuses on identifying the most frequently accessed data items, known as "hotspots" in the literature, and leverages that information to formulate a data placement problem analogous to

the graph partitioning problem in the first two systems.

2.2.1 Schism

Schism seeks to improve data partitioning for workloads that consist of simple transactions that only span a few records. The approach proposed by Schism comprises two separate phases:

- A workload-driven, graph-based replication/partitioning phase. This phase is responsible for creating a graph, each node in the graph represents either a tuple or a group of tuples, edges between nodes represent a transaction involving both nodes. After the setup, they run a graph partitioner to produce balanced partitions that minimize the number of cross partition transactions.

- An explanation and validation phase. The goal in this stage is to leverage machine learning techniques to find a predicate based explanation of the partitioning strategy, to achieve this they find a set of range predicates that perform the same partitioning scheme obtained by the graph partitioner in the previous phase.

Schism takes three different parameters as input. First, it takes a database which is being queried by the system. Second, a representative workload (e.g. an SQL trace), this input parameter provides the system with the necessary information to build the graph. Lastly, one needs to provide the system with the number of partitions desired.

After having the input workload trace the pre-processing phase begins. In this phase, the system obtains the read and write sets for each transaction present in the trace. This is done by parsing through the trace and identifying which items are requested in each transaction.

When the preprocessing is finished it is possible to create a graph that represents the database and the workload. This graph is entirely based on the information obtained from the execution traces. Upon creating the graph the next step is to apply a graph partitioner. Each tuple in the database will then be assigned to one partition while each partition is assigned one physical node.

The next stage is defined as explaining the partition. By analyzing the input trace the system seeks to collect a list of attributes which are often present in the *WHERE* clauses for each table, this is called a frequent attribute set. After having this set, the system maps rules, which are predicates on the values of the frequent attribute set, to partition numbers. This is called range-predicate partitioning. Note that to extract the rules they apply a decision tree algorithm and therefore obtain a set of rules that compactly represents the per tuple partitioning.

Lastly, there is a final validation. In this step, the system compares the cost of per-tuple partitioning, range-predicate partitioning, hash partitioning and full-table replication. The metric used is the total number of distributed transactions. The best strategy is then chosen.

As a result of executing Schism, one obtains a partitioning strategy that balances the size of the partitions while minimizing the expected price of running the workload. Note that minimizing this price requires the system to avoid cross partition transactions and as a result, the partitions, in an ideal world, comprise all the records needed for any given transaction. This is not always possible.

Schism is an offline solution and as such, does not adapt to an ever-changing workload. Furthermore, it is also important to note that by leveraging machine learning techniques to obtain a more coarse-grained mapping between data items and partitions it can deal with the scalability issues that exist when there is a one-to-one mapping between every data item and partitions. Up next we present a system that also leverages a graph structure to formulate the data placement problem but, in opposition to this work, works in an online fashion and therefore is able to adapt to changing workloads.

2.2.2 Dynastar

Scaling state machine replication has been a research topic for a while [Lam78] [Sch90], to tackle this problem a possible approach is to leverage state partitioning techniques, i.e. executing client requests on a subset of replicas. A possible solution would be a static partition scheme but due to its nature, it would not accommodate to different workloads.

Dynastar proposes a dynamic partitioning scheme. The system comprises an oracle component which maintains a global view of the workload and heuristics about data placement. This oracle allows the system to adapt to the workload submitted while minimizing state changes.

The perfect partitioning scheme would always allow commands to be executed in a single partition and therefore classic state machine replication would be applied every time within the partition. Perfect partitions would also balance the data across all of the partitions so that no partition has significant more load than others. If both of these conditions are verified then it is possible to scale state machine replication via data partitioning.

Partitioning a state machine replication service has a few challenges. The first issue is tied to the workload, it is necessary to understand it in order to achieve good partitions and that information is not always available. The next challenge is to find good partitions, this is an optimization problem that has been under research as well. Lastly, workloads vary over time due to external events which means partitions need to somehow adapt to this changing environment

in order to achieve good performance.

Dynastar allows for data to be moved between partitions, to improve system performance, in a dynamic fashion. It does not require prior knowledge about the workload, instead, it leverages an oracle that is tasked with monitoring the system and minimizing the number of state changes. This component also comprises a partitioning algorithm which allows for on-demand computation of optimized partitions. In order to perform its task, the oracle keeps two structures. The first one is a map between application state variables and partitions. The second structure is a workload graph whose vertices represent state variables and edges represent commands that comprise both variables.

In short, when a client submits a request to the system, the oracle is contacted to provide information about the location of the state variables present in the request. If the request spans multiple partitions then it is the oracle's job to choose one partition to execute the request on and communicate with other partitions in order for those to provide the execution partition with the state variables it needs. To pick a partition the oracle leverages the workload graph and the METIS partitioner [ARK06]. The metric for comparison is the number of state re-locations. The goal of this stage is to minimize this metric. The oracle is implemented as a regular tuple in Dynastar. By replicating the oracle across multiple nodes the system can tolerate failures. In order to avoid a bottleneck in the system, clients cache location information. Furthermore, clients only need to contact the oracle information the first time they access a given variable or when their cache entry becomes invalid. To tackle the issue of invalid cache entries the system simply tells the clients to retry the command.

A high-level description of the system. As input, the system will receive a client request. This request is processed by the oracle. There are three different types of requests. A client is allowed to create a variable, access a variable or delete a variable. The response from the oracle contains a set of tuples and a target partition. If it is possible to execute the command then the client simply waits for the reply from the partition sent by the oracle.

If the variables accessed by the request are in a single partition then the oracle only has to broadcast the request to that partition. Else the oracle multicasts where the request is going to be executed. This message allows for the system to gather all involved variables in the target partition selected by the oracle. Upon having all variables collected, the target partition can execute the command and send the reply back to the client. After execution, the variables return to their original sources.

Lastly, the information from the clients is leveraged by the oracle to build a workload graph. The vertices in this graph represent state variables and the edges represent dependencies between

the variables, i.e. there is an edge between two variables if there is a request that involves both of those variables. New optimized partitions are computed by the oracle periodically, and the resulting partitioning plan is then sent to all partitions which exchange variables and update states by following the plan provided by the oracle. This process does not block the execution of commands by Dynastar.

By creating the workload graph on the fly instead of relying on a workload trace, Dynastar has adaptive capabilities that Schism does not. However, the formulation of the partitioning problem is analogous as both of the systems build a graph based on knowledge about the data access patterns and transaction requests. In opposition to this, the next system addresses the data partitioning problem from a different point of view.

2.2.3 Autoplacer

The emergence of non-relational databases [DHJ+07][LM10] in a cloud computing context triggered the interest in data placement policies. The two issues a data placement algorithm needs to tackle are where to place the objects in order to maximize locality, replicating objects at the nodes where they are accessed more frequently, and minimizing lookup speed, ensuring quick access to a copy of an object.

Some systems employ the use of directories [CRS+08] [CDG+08] but, in order to support such structure, these systems only support placement at a very coarse level in order to provide placement flexibility. This tradeoff is not the only issue with directories. The use of a directory service affects system performance negatively as there is a need for additional round-trips along the critical execution path.

To tackle this issue, many state of the art systems, such as Cassandra [LM10] or Dynamo [DHJ+07], employ random placement based on consistent hashing. While this solution allows for very efficient data lookups they are not aware of access patterns to data items and therefore can produce suboptimal data placements.

Autoplacer aims to be a system that provides self-tuning data placement in a distributed key-value store. In order to achieve this goal, it leverages a self-stabilizing distributed optimization algorithm. In short, this algorithm operates in rounds and, in each round, it optimizes the placement of the objects generating the most operations.

In order to efficiently identify the hot spots, Autoplacer leverages a stream analysis algorithm. The output provided by this tool is an approximation but allows Autoplacer to track the most frequent items in an efficient manner. After obtaining these items the system can then instantiate the data placement optimization problem.

Since Autoplacer does not rely on a directory it can guarantee 1-hop routing latency. It combines consistent hashing, used as default placement strategy and employed for less popular items, with a probabilistic mapping strategy at the granularity of a data item. This design simplifies the relocation of hotspot items.

To achieve its goal, Autoplacer introduces a data structure called Probabilistic Associative Array (PAA). This structure seeks to minimize the cost of maintaining a mapping associating keys with nodes in the system. By leveraging Bloom Filters and Decision Trees classifiers Autoplacer can keep track of the elements inserted in the array in a space-efficient way and infer a compact set of rules that represents the mapping between keys and values stored in the PAA, respectively.

Periodically, Autoplacer runs a sequence of optimization rounds. If the expected gains are above a minimum threshold then a number of data items will be relocated. An optimization round comprises six tasks which are described next.

The first task involves gathering information on the top-k most accessed data items. Autoplacer only attempts to optimize the placement of hotspots so identifying those is the first step. The items marked as hotspots change in between rounds due to the previous rounds. Note that items are only relocated if the gain is still worth it. Secondly, nodes in the system will communicate with each other for statistics regarding the hotspots each of them supervises. Each node will provide the others with the hotspots it has identified in the current round. The third task has the goal of finding an optimal placement for the hotspots identified. As a result of this task, the system produces a partial relocation map. This map states where replicas of each hotspot item that the node supervises must be placed. The fourth task is responsible for encoding the relocation map into a probabilistic data structure which can be replicated at all nodes without the space concerns of the traditional map, which can potentially be very large. Each node computes the Probabilistic Associative Array (PAA) for the relocated objects it supervises. The fifth task involves broadcasting the PAA computed at each node to all other nodes in the system, Upon receiving this information each node can build a lookup table that will comprise the information on the placement of data resulting of the current round. Lastly, the data items can be moved to the new locations by leveraging state-transfer facilities in order to match the new data placement.

Autoplacer is not concerned with data consistency mechanisms used by the key-value store. The only requirement is that the consistency protocol in use must support dynamic re-adjustment of the number and placement of replicas.

In contrast to the first two systems in this section, this one does not formulate a graph

System/Trait	Operation mode	Opt, problem	Workload info	Scalability.
Schism	Offline	Graph Partitioning	Execution Traces	Machine Learning
Dynastar	Online	Graph Partitioning	Workload analysis	None
Autoplacer	Online	ILP	Workload analysis	Machine Learning
Evolve	Offline	Graph Partitioning	Symbolic Execution	Symbolic Execution

Table 2.1: Summary of data partitioning solutions

partitioning problem, instead, it employs an iterative approach to the problem via Integer Linear Programming (ILP). In similarity to the previous system, Autoplacer works online and therefore seeks to adapt to the workload it is provided with. Furthermore, Autoplacer, just like Schism, leverages machine learning and goes a step further with statistical approximations in order to avoid the one-to-one mapping which is not solved by Dynastar. Note that there is a clear trade-off between the speed of offline analysis and the ability to adapt to changing workloads.

The next section summarizes the main features provided by each of the systems presented and compares their features with the ones provided by our system.

2.2.4 Discussion

Table 2.1 summarizes the main properties of the data partitioning approaches presented and our proposed solution. In order to compare them we chose four different properties that differentiate each of the solutions presented. First, we identify their operation mode, which can be either offline or online. An offline approach does not allow for adaptations at run-time, this can result in poor partitioning decisions after the system is running for a long period of time without any changes being made to the partitioning scheme. An online solution has the downfall of requiring the system to be monitored at run time in order to identify the data access patterns required to perform data partitioning. Second, we state the optimization problem each of them formulates in order to solve the partitioning problem, the most common approaches are graph partitioners and Integer Linear Programming (ILP). The problem complexity is equivalent regardless of the formulation. Third, the method chosen to obtain knowledge on the workload the system undergoes. The data can either be extracted from execution traces provided to the system or via online analysis. Execution traces are not always available. On the other hand, online analysis might require the system to be monitored for a long period of time before all possible data access patterns are seen, especially in the presence of rare patterns Lastly, we pinpoint the techniques employed by each of the systems in order to solve the scalability constraints caused by the fact that one-to-one maps between data items and partitions rapidly explode in size and, as a result, there is often need for a technique to increase the granularity of the mapping with as minimal loss of information as possible. These techniques often involve machine learning.

After discussion, it is clear that these systems have disadvantages. Offline solutions do not adapt to the workloads they are queried with. The need for workload traces is also a handicap as it requires the system to be analyzed in run time a priori, in order to obtain a data partitioning scheme. Lastly, it is necessary to generalize the mapping between data items and partitions with minimal loss of knowledge. A solution that does not tackle this issue will not scale. In the following section, we present our system that seeks to leverage symbolic execution to solve the data partitioning problem whilst avoiding any of the downfalls identified in other systems.

Chapter 3

Evolve

Distributed transactions impose a large penalty to the performance of modern databases due to the high cost inherent to distributed protocols. A good data partitioning strategy can minimize the occurrence of these transactions by ensuring that the data a transaction requires in order to execute is present within a single partition. Evolve presents an approach to database partitioning that leverages symbolic analysis of transaction code with the purpose of minimizing the number of transactions that span multiple partitions.

In this section we will start by presenting the system overview. We will follow a top down look approach to provide the reader an idea of what is done in each step in order to reach the final result. We will then describe each component of our system in further detail.

Figure 3.1 shows the four different components of our solution, each working on the output provided by the previous stage.

First, the symbolic analysis stage is responsible for analyzing the code of each transaction[1] via symbolic execution. The result of this step is a set of symbolic formulas representing the data accesses of each unique code path in a transaction. Collectively, we call those formulas a transaction profile.

Second, the symbolic formulas, representing the transaction profiles, are passed to a graph initialization component. In this stage, the various transaction profiles are mapped into symbolic vertices. Each vertex will be representing, symbolically, a unique set of data items. An edge will represent a transaction that accesses data represented in both vertices.

Third, the vertices undergo a splitting process to prepare the graph for the next stage, graph partitioning. Splitting is necessary in most cases as the initial graph may contain vertices of very different sizes, which will not allow for useful partitioning. Moreover, the graph may not even contain enough vertices to perform partitioning in case there are very few transactions.

[1] this can be executable or the source code

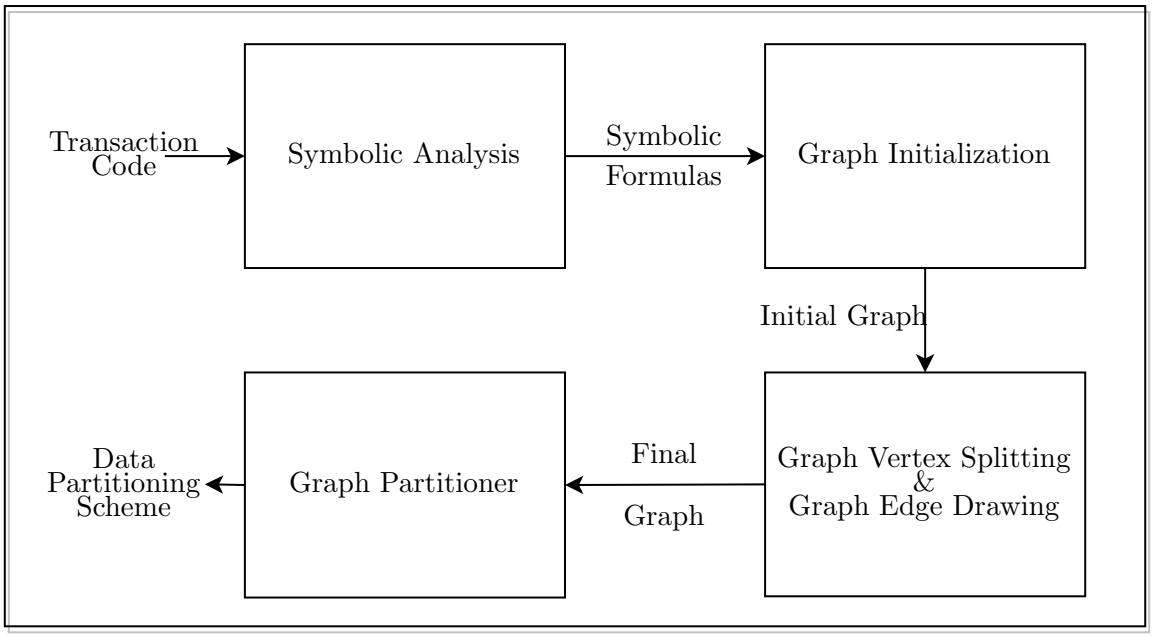

Figure 3.1: System overview

The splitting phase has the purpose of increasing the number of vertices whilst minimizing the negative impact of decoupling data access that belong to the same transaction profile i.e., leading to more edges in the graph. In this stage we also compute the edges present in the final graph.

Finally, we perform graph partitioning on the graph resulting from the aforementioned splitting process. For this purpose we leverage a graph partitioner that outputs a partition for each vertex according to the graph and balance constraints provided. As a result, at the end of this pipeline we have a set of symbolic vertices each mapped to a single partition. Since these symbolic vertices can be translated into data items, this provides us with a database partitioning scheme. We will now describe each component in further details.

3.1 Symbolic Analysis

State of the art database partitioning solutions such as Schism[CJZM10] and Dynastar[LFEK$^+$19] leverage execution traces in order to extract information about transaction profiles, namely the data items accessed by a given transaction. In our solution, instead we propose a different approach that extracts the transactions profiles from the application code using symbolic execution. These profiles define, in a symbolic way, the read- and write-sets of a transaction: data items read and written by a transaction.

Each transaction is composed of a set of unique execution paths. Each of these paths is defined by a set of conditions on the transaction input and state of the data. When an instance of a transaction is executed, one of those paths will be followed accessing a set of data items.

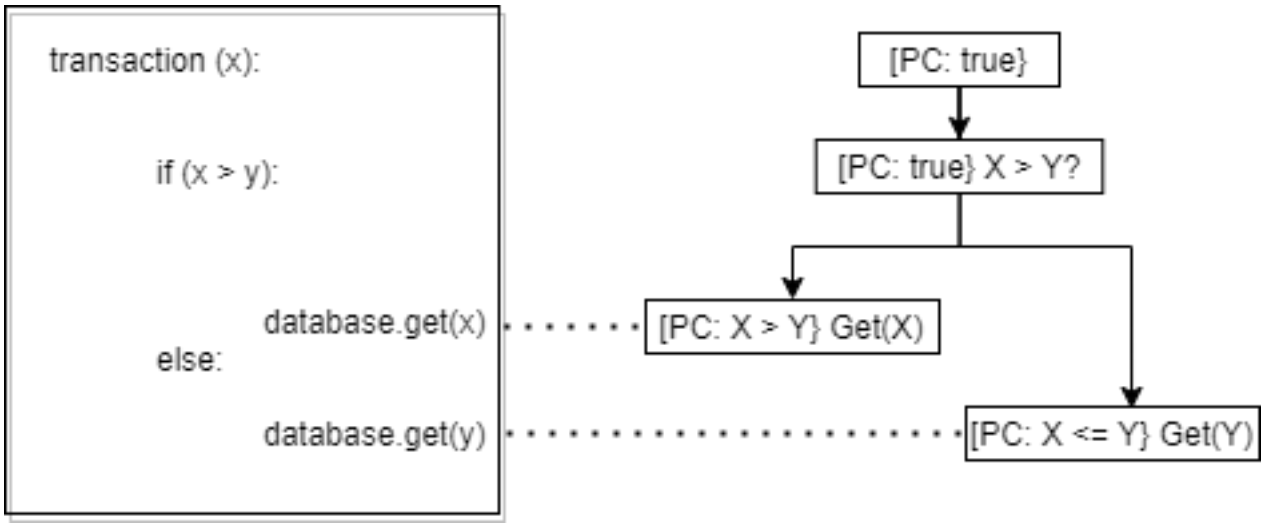

Figure 3.2: Symbolic execution example. PC stands for Path Condition

We define a transaction profile as tuples, with the first entry being a path constraint (the set of conditions on input and state of the data) and the second being the set of data accesses of that path, for all the transactions' execution paths. Figure 3.2 illustrates these concepts. Note that in this example, X is a transaction input while Y is a value stored in the database. There are two possible execution paths each corresponding to a different path condition, (PC). Lastly, each path leads to a different database access. If X is greater than Y then the accessed item will be X otherwise Y is accessed.

Our solution utilizes the Java Path Finder (JPF), a well established symbolic execution engine developed by NASA for Java programs. One of the advantages of JPF, and the main reason behind adopting it is its listeners API. This allows the programmer to invoke custom code upon relevant events, such as instruction execution, function call, function return, conditional statements, amongst other. In our case, we need to intercept database operations. Currently, we assume a key/value store model and thus there are two operations we are interested in:

- GET operations corresponding to values being read which modify the read-set of a transaction profile
- PUT operations, which update or insert an item (if it does not exist), and, as such, modify the transaction's write-set.

When either of these operations is invoked, we extract the symbolic formula identifying the key that is either being read or written and add it to the read- or write-set of the transaction profile under the path constraint of the current execution path.

As mentioned above, the execution paths of any transaction are dependent on two factors. The input given to the transaction and the state of data items read during transaction execution.

We define transactions that do not rely on database state as direct and those that do rely on database state, such as the value of data items read, as indirect. The example in Figure 3.2 presents an example of an indirect transaction as the value of Y relies on the state of the database. A sample of a direct transaction can be created by replacing Y with any concrete value.

As a result, the symbolic formulas can either state that a data item will be accessed directly or indirectly. For the latter, a transaction needs to read a value from the database to determine which item will be accessed. This adds complexity as the state of the database can affect execution paths, In order to deal with these cases,as when the symbolic engine comes across a conditional branch it has to check whether the symbolic formula that will be added to the path is indirect, i.e. relies on database state and, in that case, the data items need to be extracted.

In order to guide the symbolic execution engine towards our goals, we took the following design decisions. First, the system performs a depth first exploration of the execution paths, allowing it to discard states which do not lead to any new information and therefore reducing the amount of memory used by the symbolic execution engine. Second, it leverages a tree structure in which the nodes express a given path and comprise the set of symbolic formulas collected between the node's execution path and the next conditional statement providing a convenient format.

In more detail, each path is explored by executing a transaction symbolically until a conditional statement is reached, collecting information about any data items read and written in between. The list of items accessed is stored within a node alongside the logical condition, all in symbolic form. The node will then have two sub-trees corresponding to the different outcomes of the conditional statement. The tool then performs a depth first search and, as a result, upon reaching the end of the program for a given execution path, it stores the information regarding the path constraints and corresponding read- and write-sets followed by a backtrack in the tree to explore other execution paths. If two children sub-trees map to the same read- and write-sets they are pruned and their read and write set is added to the parent node. This optimization allows for the reduction of the memory consumption during symbolic analysis, the same would not be possible for a breadth first search, before exploring all the execution paths. As a consequence, the tree resulting from executing symbolic analysis on the code of a given transaction is compact and efficient to query at run time. Note that predicting the accesses of a transaction at run time can be done in logarithmic time by exploring the tree and its appropriate path.

The description so far leverages previous work, it was included in this section for completeness as it is an important component of Evolve. Next we describe the improvements done in the scope

of this thesis in regards to the symbolic analysis tool.

A known problem with symbolic execution is path explosion: a single loop with N iterations containing one conditional statement generates up to 2^N execution paths. In order to tackle this issue we developed an optimization. This optimization consists of executing a preliminary static code analysis phase in which we identify irrelevant variables. By using static analysis tools like Soot[VRCG+10] we can identify those variables that do not affect the read and write set of a given transaction profile, either explicitly via direct assignment or implicitly via information flow. During the execution of symbolic analysis we mark these variables as concrete and assign them a value. This allows for certain conditional statements to rely solely on concrete variables and, in that case, only one execution path needs to be explored which greatly reduces the amount of paths that need to be analyzed.

At the end of this first stage, we have the sets of symbolic formulas per transaction profile. These formulas identify the read and write sets of the transaction for any possible inputs and database states. An example of a symbolic formula:

$$2- > warehouseid + (districtid * 10)$$

In the formula above, the prefix represents the identifier of the table being accessed (2) while the expression itself evaluates to data item identifiers. Assuming *warehouseid* $\in$ [0,1] and *districtid* $\in$ [0,1] then the data items accessed by this particular symbolic formula would be $\{0, 1, 10, 11\}$. More clarifications on these formulas will be provided in the next section. The sets of symbolic formulas are then sent to the graph initialization module which we describe next.

3.2 Graph Initialization

In this stage, the goal is to build a graph by leveraging the information provided by symbolic analysis. This section starts by explaining the structure of the graph and proceeds to explain how we encode the symbolic analysis information into a graph.

Each vertex in the graph will contain a set of data items. An edge exists between two vertices if there is a transaction that accesses data on both vertices. By design, the vertices in the graph are disjoint, i.e. there will not be a data item present in two different vertices. However, the process that ensures that vertices are disjoint is only performed in the next stage, after each vertex undergoes splitting.

Two other relevant concepts are the weight of vertices and edges. The weight of a vertex

is equal to how many data items it contains. The weight of an edge corresponds to how many transaction instances access data on both vertices, i.e. how many inputs cause a given transaction to perform remote accesses. This concept is clarified later on.

The input received by the graph initializer is a set of transaction profiles each comprising a set of symbolic formulas that identify the data accesses for their respective transactions.

In order to encode symbolic information on the vertices we resort to two definitions. Each symbolic formula identifying a set of data items is defined as ρ while the range of its input variables is defined as ϕ. Each ρ always has an associated ϕ. Expression 3.1 illustrates an example. The prefix within the ρ expression identifies which table is being accessed, i.e. table two in this case. The input variables for that particular ρ are *warehouseid* and *districtid* whose range is given by the respective ϕ.

$$\rho : 2- > warehouseid + districtid * 10, \phi : warehouseid \in [0,1], districtid \in [0,9] \quad (3.1)$$

Moreover, it is possible to provide a symbolic formula that contains two different ρ formulas, each of them with an associated probability. This is particularly useful in scenarios where, in addition to the information obtained through symbolic analysis, there is additional knowledge on the domain of the problem. We define a ρ with an associated probability as a probabilistic ρ.

Each transaction profile identified during the symbolic analysis stage will generate a new vertex in the graph. Furthermore, vertices store a set of ρ and ϕ, each of the elements corresponding to a set of data items. We define the union between all those sets stored within a vertex as σ.

The initialization process is as follows. Each transaction profile identified by the symbolic analysis is parsed through. This profile will contain a set of ρ and associated ϕ. The first step is to analyze each ρ and check whether the symbolic formula is direct or indirect. From the previous section, direct formulas rely solely on the values of the input variables whilst indirect ones rely on the database state.

Expression 3.2 presents an example of an indirect formula. If the ρ formula is indirect then it needs further processing. In this example the underlined operand is an indirect read and its value depends on the state of the database.

$$\rho : 6- > (district_id + \underline{((GET(2- > (district_id + (warehouse_id * 100)) - > 10)} * 10000))) \quad (3.2)$$

We solve this problem by translating what we call an indirect read into an extra input variable whose range is aligned with the problem's description, ensuring correctness. More precisely, the range of this variable is obtained through information regarding the database schema and transaction descriptions. Note that this can be done automatically during graph initialization. Expression 3.3 shows the updated version of the ρ formula.

$$\rho : 6- > (district_id + \underline{indirectorderid} * 10000))) \tag{3.3}$$

In this second expression, the operand whose value relied on a value read from the database no longer exists, instead there is a placeholder variable. Once again, to ensure correctness, the range of this variable will be an over approximation in order to cover all possible values that could be read from the database by the initial formula.

Once there are no more ρ formulas containing indirect reads we can generate the σ for the transaction profile under analysis, completing the creation of one graph vertex. This process needs to be performed for all transaction profiles to finalize the graph initialization process.

At this stage, we have a set of graph vertices not connected to each other. Each vertex stores a set of data items that can have overlapping data items with other vertices. This fully disconnected graph is now ready for the next stage in which each vertex will undergo splitting followed by a process that ensures vertex disjointedness and computes the edges

3.3 Graph Vertex Splitting

This stage comprises a sequence of steps that transform a fully disconnected graph received as input into a graph ready to be partitioned. Each vertex in the initial graph is analyzed and undergoes a process in which it is split into a set of disjoint sub-vertices whose union is equal to the original vertex. After being split the sub-vertices are then added to the final graph. During this step we force vertices to be disjoint, i.e. there can not be a single data item present in two different vertices. The edges in the final graph are also drawn in this stage. Finally, the graph will be ready for partitioning and will be fed into the next stage. We detail the entire process.

3.3.1 Reasons for Splitting

The ultimate goal of this stage is to split a vertex into a set of sub-vertices while maximizing the use of information obtained through symbolic analysis. It would be rather trivial to keep the graph as is and not perform any splitting. This would mean that all the items needed by any transaction are contained within a single vertex - ensured by symbolic analysis identifying the

$$1- > warehouseid$$
$$2- > (districtid + (warehouseid * 100))$$
$$3- > (districtid + (warehouseid * 100) + (customerid * 10000))$$
$$4- > (districtid + (warehouseid * 100) + (customerid * 10000))$$

Figure 3.3: Symbolic formulas for the payment transaction in TPC-C

accesses done by a transaction and us placing them into a single vertex in the previous stage.

However, this approach would mean that we would have as many vertices as transaction profiles and the number of possible partitions and respective balance would be very limited. For example, the TPC-C benchmark has only four different update transactions, if we followed the approach aforementioned we would not take any advantage of a system with, for example, ten available machines as we would lack vertices to cover all partitions. Moreover, it is likely that different transaction profiles impose different loads on the system and, for balancing purposes, it is much more desirable to have the possibility of partitions covering more than one transaction profile.

To avoid these limitations we perform vertex splitting. In order to maximize the use of symbolic analysis whilst still generating a better graph for partitioning we define a metric to follow during splitting - we want to minimize the resulting edge weight from splitting a given vertex.

Recall that an edge exists between two vertices when a transaction instance accesses both vertices and that the weight of an edge is equal to how many transaction instances access data on both vertices. Splitting a vertex can easily generate an edge as it can decouple transaction instances that were initially grouped together. Note that splitting a vertex is no more than creating two or more disjoint subsets of data items whose union is equal to the original vertex.

3.3.2 Splitting approach

We will now detail the splitting process. In order to provide the reader with a clear explanation of how the whole process develops we will often recur to small examples based on the TPC-C benchmark.

Initially, each vertex has an associated σ containing a set of ρ formulas and respective ϕ. These formulas will span a multitude of tables depending on the transaction profile. Figure 3.3 presents the formulas obtained for the payment transaction profile in the TPC-C benchmark. In this example we can see four different ρ formulas. There are also four different tables being accessed identified by the prefixes in each of the formulas.

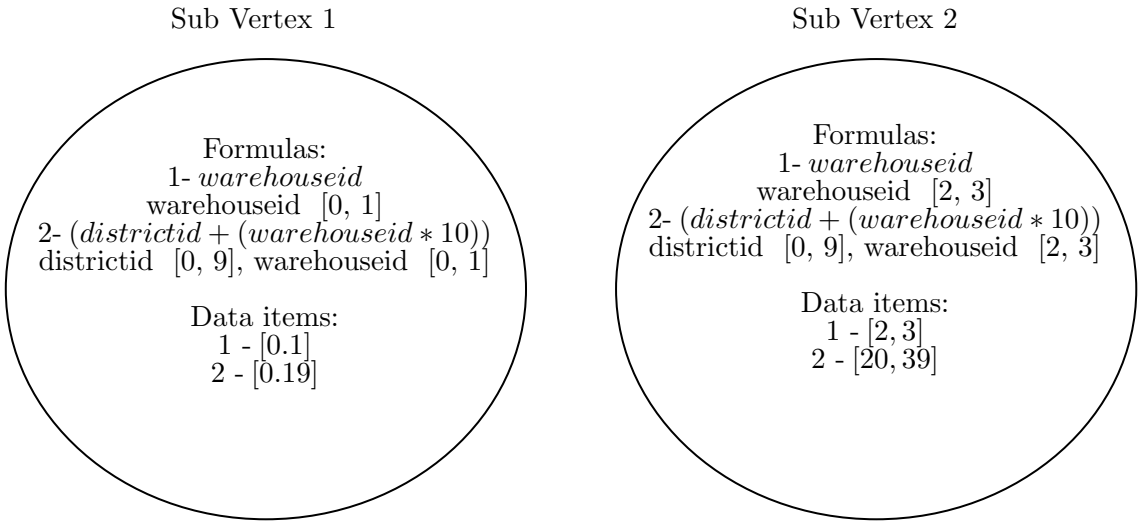

Figure 3.4: Vertices resulting from a split by *warehouseid*

The first step is to group ρ formulas according to which table they access. In this example, all the formulas access a different table but that is not always the case. We denote the structures that hold all the formulas related to a given table as buckets. These buckets allow us to isolate each table and treat each of them individually going forward.

It is now important to reason on how to split a vertex. A first idea could be to split each different ρ formula into a different sub-vertex. This idea, while simple, does not align with the metric we have set. Based on the example in Figure 3.3, if we chose to perform this split we would have four disjoint sub-vertices, each containing a single ρ formula. Furthermore, any payment transaction instance would require access to four different vertices in order to collect all the data it needs to execute. According to our edge definition this would mean all the sub-vertices resulting from the split would have edges connecting them. Given that we want to minimize the amount of edges and respective weights upon vertex splitting this does not work for us. In fact, it corresponds to the worst case scenario in which all transaction instances require remote accesses.

Another possible approach is to reason on splits based on partitioning the input range of the symbolic variables in the ρ formulas. Recall that each ρ always has an associated ϕ, performing this sort of split would simply mean constraining the values that one or more of the symbolic variables can have in each of the sub-vertices. This can be achieved by manipulating the ϕ expressions.

For the example in figure 3.3, consider a split in two distinct sub-vertices by splitting the original vertex based on the range of the warehouseid variable. Let us assume warehouseid $\in$ [0, 3] originally. A possible split would be to assign warehouseid $\in$ [0,1] to one sub-vertex and warehouseid $\in$ [2, 3] to the other sub-vertex. Figure 3.4 shows this split, for simplicity we will

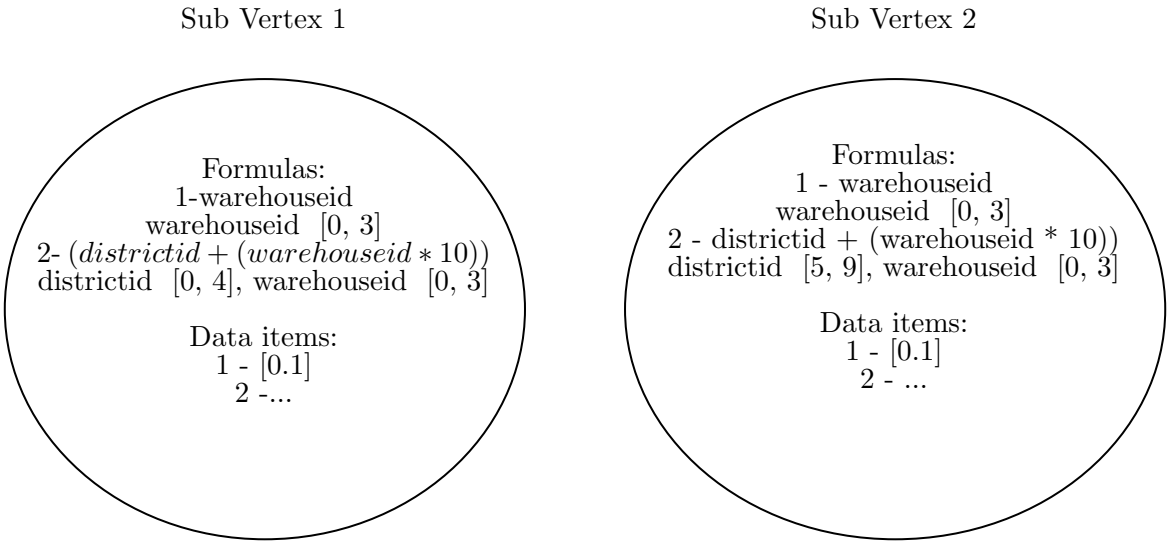

Figure 3.5: Vertices resulting from a split by *districtid*

only show what happens across the first two tables however the same process could be extended to the others.

Let us now analyze what happens with the edges when we perform this split. The figure illustrates the ρ formulas and respective ϕ after a split by the *warehouseid* variable. It also shows which data items are stored within each of the vertices for both table one and table two. These sets, specified on the bottom of each vertex do not overlap, i.e. the set of data item identifiers in each are disjoint, which means that there is no need to draw an edge between the two resulting sub-vertices. Note that our vertices are going to be disjoint, i.e. have zero overlapping data items, in a step posterior to the splitting phase. Any overlaps caused by splitting will later turn into edges, hence the need to minimize the overlaps during the splitting stage. This conversion between vertex overlaps and graph edges will be further clarified in the subsequent stages.

We can then conclude that this split resulted in additional edge weight of zero which is the optimal scenario given our metric. An interesting idea would be to repeat the same splitting process but considering the *districtid* variable instead. Figure 3.5 illustrates this split.

A very simple analysis, based solely on the first table, shows that this split causes the sets of data items represented by each sub-vertex to overlap. This split will therefore result in additional edges which means it is a worse split than the previous one, by *warehouseid*, which we found to be optimal. This leads us to another important conclusion. The variable which we pick for splitting a vertex matters: certain variables will yield better results than others depending on the transaction profile.

Picking the correct splitting variable is then the next problem we have to solve. Understanding why *warehouseid* worked better than *districtiid* in the examples above is the first step. Let

us introduce the concept of variable coverage. We define the coverage of a variable as the number of ρ formulas it appears in for a given transaction profile. For the example above, *warehouseid* has a coverage of four while *districtid* only achieves a coverage of three.

The fact that *districtid* does not cover the first table, i.e. it is not present in any formula accessing table one, ensures that any split using solely this variable will not be able to split this table. As a consequence, any set of sub-vertices created by the split will have overlapping data items as the entirety of table one is present across all sub-vertices. This leads us to a very important conclusion. In order to minimize the resulting edge weight from splitting it is mandatory to have all ρ formulas present in the transaction profile covered by the set of splitting variables.

There might be scenarios in which there is no variable that achieves full coverage, i.e. the variable coverage is not equal to the amount of ρ formulas present in the transaction profile, in which case it is necessary to pick more than one splitting variable. Note that failing to achieve this full coverage means that there will be a table that is fully overlapping across all sub-vertices resultant from splitting and, as mentioned above, all these overlaps will become edges in a subsequent stage meaning the split would be bad according to our metric that seeks to minimize edges and corresponding weights.

Now that we have discussed how to split a vertex whilst minimizing the resulting edge weight from splitting we are ready to proceed with our example based on the TPC-C benchmark.

Recall that we had placed all ρ formulas into buckets corresponding to all the different tables in the transaction profile. The next step is to analyze each table and decide how to split the vertex. Algorithm 1 provides an overview of the process. The purpose of this algorithm is to collect a set of variables that can possibly split the vertex. During this stage we will create a

list of possible splitting variables that we will pick from in the subsequent stage.

Algorithm 1: Table Analysis

Data: Bucket

Result: List of candidate splitting variables for each table

```
1  candidateSplits ← newList();
2  if bucket.size() == 1 then
3      candidateSplits.add(bucket[0].getVariables());
4      return candidateSplits;
5  end
6  for i ← 0; i < bucket.size(); i++ do
7      ρ1 ← bucket[i];
8      for j ← i + 1; j < bucket.size(); j++ do
9          ρ2 ← bucket[j];
10         intersection ← computeIntersection(ρ1, ρ2);
11         if ¬intersection then
12             candidateSplits ← candidateSplits ∪ ρ1.getVariables();
13             candidateSplits ← candidateSplits ∪ ρ2.getVariables();
14         else
15             commonSplitters ← checkCommonSplitters(bucket);
16             if #commonSplitters != 0 then
17                 candidateSplits.addAll(commonSplitters);
18                 return candidateSplits;
19             else
20                 if replicationOption then
21                     candidateSplits.removeAll();
22                     markTableForReplication();
23                     return;
24                 else
25                     candiateSplits.removeAll();
26                     markTableForTableSplit();
27                     return;
28                 end
29             end
30         end
31     end
32 end
33
```

$$FindInstance[\rho 1 == \rho 2 \,\&\&\, \phi 1 \,\&\&\, \phi 2,\, \{\{vars\, in\, \rho 1\},\, \{vars\, in\, \rho 2\}\},\, Integers])$$

$$\begin{aligned} FindInstance[warehouseid1 == districtid2 + (warehouseid2 * 100) \\ \&\&0 <= warehouseid1 <= 3 \\ \&\&0 <= warehouseid2 <= 3 \\ \&\&0 <= districtid2 <= 9, \\ warehouseid1, warehouseid2, districtid2, Integers]) \end{aligned}$$

Figure 3.6: Structure and example of Mathematica *FindInstance* query

Initially, we iterate over all the ρ formulas for a given table. If there is only one formula for a given table then all of those variables are possible splitters and, more importantly, at least one of them must be picked to achieve full coverage of the ρ formulas. If there is more than one formula in any bucket then we need further analysis.

In the case that the formulas within the same bucket do not intersect, i.e. they reference fully disjoint set of data items, then the algorithm simply adds the variables in each ρ formula to the list of possible splitters for that specific formula and ensures at least one variable is picked in the subsequent stage.

In order to compute whether the formulas overlap or not we utilize Wolfram Mathematica[Wol]. We build a FindInstance query which will have as operands both ρ formulas under analysis and their respective ϕ. Figure 3.6 provides the structure of these queries and an example for the TPC-C benchmark.

At this point we come across another problem. If there are more than two ρ formulas in a given bucket there is a possibility that they overlap between themselves, i.e. the set of data items referenced by both is not disjoint, this means that simply picking a splitting variable that achieves full coverage might not be sufficient. The decoupling inherent to splitting might originate data overlaps between vertices. In order to avoid the occurrence of these overlaps it is necessary to ensure these data items are mapped to a single vertex during splitting. Figure 3.7 presents an example of two formulas that are partially overlapping. In particular, they overlap when *warehouseid1* is 0 and *warehouseid2* is 3.

In this example the formulas have a variable in common, *warehouseid*. We denote this variable as a possible common splitter. Note that, in the general case, there could be more than one common splitter. We will then simulate a split of the overlapping formulas based on each of the common splitters identified. For the split to be successful it needs to keep the overlapping items in one of the parts, i.e. avoiding duplicated items across both splitted parts. Figure 3.8

$$FindInstance[warehouseid1 + 3 == warehouseid2)$$
$$\&\&0 <= warehouseid1 <= 3$$
$$\&\&0 <= warehouseid2 <= 3$$
$$warehouseid1, warehouseid2, Integers])$$

Figure 3.7: Example of overlapping formulas

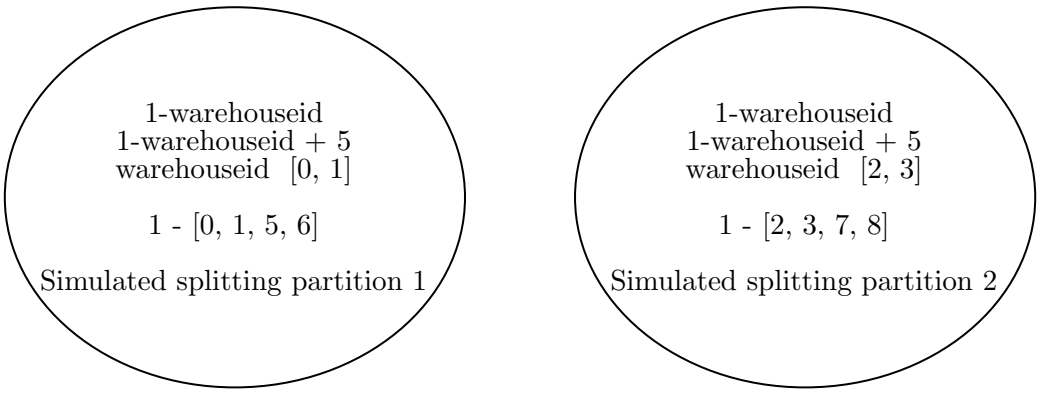

Figure 3.8: Outcome of a simulated split on overlapping formulas

illustrates the result of a simulated split given the formulas in Figure 3.7.

In this particular scenario, the simulated splitting partitions do not overlap. This allows us to add the common splitter identified, *warehouseid*, to the list of splitting variables. On the other hand, in case a valid common splitter is not found then we proceed with the analysis.

To deal with this situation we provide two different solutions. The first one is based on replication in which we allow the table that contains overlapping formulas to be replicated across the sub-vertices. The second one introduces a concept which we denote as table splitting. We will now describe both approaches.

Adopting replication is simply allowing for data items to be present in two or more different vertices. This goes against the design choice that we have set of having fully disjoint vertices. However, this is an extreme case in which there is no perfect way of splitting a given table and as such replicating it might be the better solution depending on a variety of factors such as transaction frequency. Recall that we are yet to ensure vertex disjointedness, this will only be done in a subsequent step. Following the replication approach at this stage would simply marking certain ρ formulas as replicated allowing them to have overlaps later on the pipeline.

Allowing for items to be replicated across all the sub-vertices can impose high costs to the database performance, depending on whether these data items are frequently updated. Therefore we propose table splits. As an alternative, in this case, instead of splitting a vertex by constraining the ranges of the variables in the ρ formulas via manipulating its associated ϕ,

as we do for the regular splits, we constrain the value that each ρ formula can yield directly. Expression 3.4 presents an example of a table split. In this example, the value of the formula is capped at 250, allowing us to split the table in half. The first half is composed by data item identifiers below 250 and the second half has the remaining items.

$$districtid + (warehouseid * 100) \underline{\leq 250} \tag{3.4}$$

This approach allows us to group together any overlapping data items between ρ formulas on the same table. However, this approach has the downside of not providing balance between the vertices resulting from splitting. Introducing a constraint on the values that a given ρ formula can have might result in one of the sub-vertices having a higher number of data items than another one. To mitigate this effect, the threshold used for table splits must be carefully selected taking into account how many sub-vertices we desire to achieve from splitting a given table and, most importantly, what are the values the data item identifiers can have in said table. Note that a ρ formula is nothing more than a function whose output is a data item identifier. Domain knowledge on the data item identifiers can easily be obtained through the database schema.

Figure 3.9 presents an overview of this process. In short, if there are no overlaps between the ρ formulas for a given table then all variables present across the formulas are candidate splitters, in this case we apply splits based on partitioning the input range of the variables in each formula. In the occurrence of overlaps between two ρ formulas on the same bucket, i.e. on the same table, we check whether we can find a splitting variable that deals with the overlap, a common splitter, if we can not find a variable with such properties we either replicate the table across all sub-vertices resulting from splitting or we perform a table split ensuring the overlapping data items are mapped together during splitting. After collecting the candidate splitters we are ready to proceed.

3.3.3 Selection of splitting variables

The next step is to select within the list of candidate splitters which variables to select to split each vertex. Recall that in order to minimize the resulting edge weight all the ρ formulas must be split by at least one variable, unless that formula belongs to a table which requires either a table split or replication. The tables that will undergo a table split or replication are, for the purpose of the next stage, already dealt with, i.e. they are not involved in the selection of splitting variables. The ρ formulas that belong to any other table will be under analysis.

Initially, we build an array with as many entries as ρ formulas across the tables that need

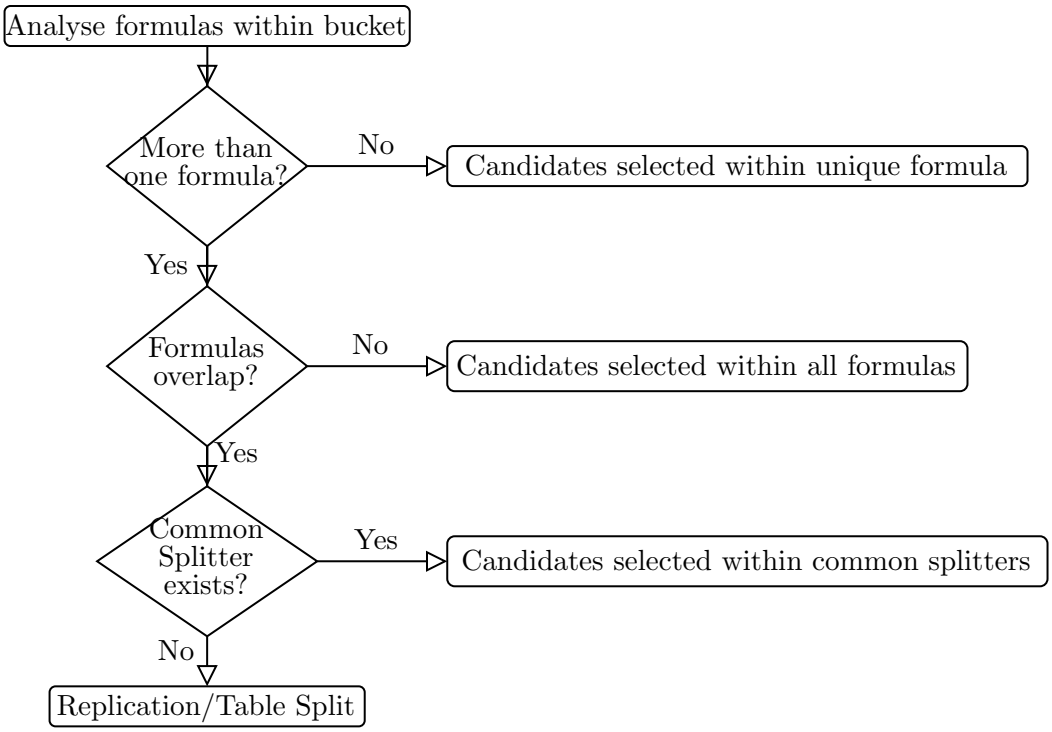

Figure 3.9: Table splitting approach overview

to be covered by a splitting variable. Each entry in this array will contain the list of candidate variables, collected in the previous stage, that can split the ρ formula. Note that this process happens for every vertex, i.e. every transaction profile, independently just like the splitting process itself.

We have discussed earlier that certain variables result in better splitting than others, we have also concluded that this is directly related to the coverage of each variable, that analysis was based on Figure 3.5. Let us now discuss how to pick the ideal set of splitting variables.

Assume a given vertex can be split solely by one splitting variable which achieves full coverage. In this scenario, we are aligned with our TPC-C example in which *warehouseid* achieves full coverage of the ρ formulas in the payment transaction profile. This allows us to split the vertex without generating any extra edges resulting from splitting and, as a result, we have the optimal splitting solution according to our metric that seeks to minimize the resulting edge weight caused by splitting. We have also concluded earlier that if a variable does not achieve full coverage and it is picked as a splitter it will generate additional edges.

There is also the possibility that no single variable achieves full coverage, in which case more than one splitting variable has to be selected. Recall that if any ρ formula is left uncovered by the set of splitting variables it will definitely generate edges. Consequently, it is necessary to pick the set of splitting variables that achieves full coverage cautiously.

If no single variable archives full coverage then any variable added to the set of splitting

$1 -> warehouseid, \{warehouseid\}$
$2 -> (districtid + (warehouseid * 100)), \{warehouseid, districtid\}$
$3 -> (districtid + (warehouseid * 100) + (customerid * 10000)), \{warehouseid, districtid, customerid\}$
$4 -> (districtid + (warehouseid * 100) + (customerid * 10000)), \{warehouseid, districtid, customerid\}$
$Hitting - Set : \{warehouseid\}$

Figure 3.10: Candidate splitters for each formula and corresponding final splitting variables

variables will increase the edge weight as it will not be able to cover, at least, one ρ formula. This is what happened in the TPC-C example in Figure3.5 when we attempted a split by *districtid*.This remark is very important as it forces us to select the smallest set of splitting variables that achieves full coverage in order to minimize the resulting edge weight.

Therefore, the ideal set of spitting variables is the smallest set of variables that achieves full coverage. Recall that we have a list that stores in each entry which variables split the associated ρ formula. The problem is then simply picking the smallest set of variables that intersects with at least one element in each entry on our list. This is a NP-Complete problem known as hitting set. However, each vertex, corresponding to a transaction profile, is analyzed individually and, as a result, the number of variables present across the ρ formulas will be, in most cases, very small. Therefore, for the scope of our problem, the computation of the hitting set is trivial and does not impose any additional obstacles.

After computing the hitting set, we have the smallest set of variables that splits a vertex. Possibly, we will also have a set of tables that needs to undergo a table split or replication discussed previously. Note that both alternatives are mutually exclusive, only one of these approaches is applied to a given table. Figure 3.10 illustrates the process for the TPC-C example we have been studying, each formula provides a set of candidate splits, the hitting set computed identifies the split we select. Note that this is a trivial example without any overlaps and, as a result, the list of candidates is equal to the list of variables in each formula. We are now ready to move on and apply the selected splits to the vertex.

3.3.4 Applying the splits

At this stage, we have computed how to split each vertex, the next step is to apply the splitting logic to each vertex and generate all the sub-vertices. It is important to note that for every split applied to a vertex, may it be a split based on the set of splitting variables or a table split, it affects the number of sub-vertices that need to be created.

Having the splitting variables identified, it is important now to select a splitting function to

apply to the vertex in order to generate sub-vertices. In the example we have been using, as well as in our implementation, we decided to split the vertices by slicing the range of the input variables given to the ρ formulas uniformly, by manipulating the ϕ associated with each ρ.

However, we provide an interface that allows for the introduction of any splitting function. This enables the exploration of more complex splitting techniques. Moreover, the number of input partitions generated by each variable and, consequently, the number of sub-vertices created is also adjustable.

It is important to mention that the number of sub-vertices generated depends on a few different factors. First, it depends on the splitting function being used. If the splitting function is simply slicing the range of the input variables then the number of sub-vertices relies on the number of slices generated and the number of variables in the set of splitting variables. Expression 3.5 presents the formula that computes the number of sub-vertices resulting from splitting in this scenario. Note that this formula can be extended to include table splits. For that purpose, a table split is no more than another variable that is splitting a table in slices as well.

$$\#vertices = \#slicesbyvar1 * \#slicesbyvar2 * ... * \#slicesbyvarN \tag{3.5}$$

After applying the splits to a vertex, we end up with a set of sub-vertices that may or may not have items in common. This depends on whether there was a splitting variable that achieved full coverage or we had to resort to a set of splitting variables instead. As previously mentioned, any duplicated data items across vertices will result in edges, the purpose of the next stage is to identify these overlaps between vertices and draw the respective edges.

3.4 Graph Edge Drawing

At this point, we have a set of vertices that were created during splitting, these vertices will now be added to the graph, one by one. Every time a vertex is added to the graph it needs to be compared to all previously existing vertices in the graph in order to ensure that there are no duplicated data items. Every time a collision is found between a newly added vertex and an already existing vertex, the data items that are duplicate are removed from the new vertex and an edge is created between the two vertices. This process lasts until all vertices are added to the graph. Algorithm 2 illustrates this process.

The algorithm starts by selecting each vertex and comparing it to any vertex already present in the graph. As a result, the first vertex that is added to the graph does not need any com-

parisons, however every subsequent vertex needs to be compared to all pre-existing ones. To compare a vertex to another it is necessary to compare each ρ formula and associated ϕ in the first vertex with all of the ρ formulas and associated ϕ on the second vertex. As an optimization, it is only necessary to compare ρ formulas to those that access the same table which greatly improves the process. This is due to the fact that we are only interested in finding data item overlaps and different tables store different items by design.

Algorithm 2: Graph edge detection process

```
Data: newVertex, graph
Result: Graph with the extra vertex added and respective edges
1  rhosNewVertex ← newVertex.getRhos();
2  foreach rho formula ρ ∈ rhosNewVertex do
3      ρ1 ←ρ;
4      ϕ1 ←ρ1.getPhi();
5      foreach graph vertex gv ∈ graph do
6          rhosGraphVertex ← gv.getRhos();
7          foreach rho formula ρGV ∈ rhosGraphVertex do
8              ρ2 ←ρGV;
9              ϕ2 ←ρ2.getPhi();
10             intersection = computeIntersection(ρ1, ϕ1, ρ2, ϕ2=;
11             if intersection == {} then
12                 continue;
13             else
14                 ρ1.performSubtraction(intersection);
15                 newEdge ← createEdge(newVertex, gv, intersection);
16                 graph.addEdge(newEdge);
17             end
18         end
19     end
20 end
21 newVertex,computeVertexWeight();
22 graph.addVertex(newVertex);
23 return graph;
24
```

WIth the objective of detecting overlaps between ρ formulas across different vertices we deploy Wolfram Mathematica once more. We use *Reduce* queries for this purpose. An example

$$Reduce[\rho 1 == \rho 2 \,\&\&\, \phi 1 \,\&\&\, \phi 2,\ \{\{vars\, in\, \rho 1\},\ \{vars\, in\, \rho 2\}\},\ Integers])$$

$$\begin{aligned} Reduce[warehouseid1 == districtid2 + (warehouseid2 * 100) \\ \&\& 0 <= warehouseid1 <= 3 \\ \&\& 0 <= warehouseid2 <= 3 \\ \&\& 0 <= districtid2 <= 9, \\ warehouseid1, warehouseid2, districtid2, Integers]) \end{aligned}$$

Figure 3.11: Structure and example of Mathematica *Reduce* query

of this query is present in Figure 3.11. The operands are, once more, the ρ formulas and associated ϕ.

These queries output as a result an expression that specifies which data items are in common between both vertices, if any. Note that the output of this query is expressed symbolically, i.e. as an expression containing constraints on the input variables in each ρ formula. This allows us to concatenate the result of the Mathematica queries into our ρ formulas, subtracting the overlapping set of data items without needing to fully specify which data items are being removed, preserving our symbolic representations.

It is important to note that we only update the ρ formula of the vertex that is currently being added to the graph. The vertex that was already present in the graph does not suffer any changes. This design decision allows us to add new vertices to the graph without needing to perform alterations to all previously existing ones upon each addition.

After detecting a collision between two vertices and performing the subtraction process, the next step is to compute the edge resulting from removing a set of data items from the new vertex. Note that without any subtractions each vertex would map to a subset of a transaction profile as vertices were created directly from transaction profiles obtained through symbolic execution and then split into subsets during vertex splitting. Removing the overlapping data items from a vertex will cause the transaction instances it contained to not have all the data times they require. These missing items will be present in the vertex that was overlapping with the current one. Therefore, every time a collision is detected between the latest added vertex and an existing one, we can draw an edge between the two vertices identifying that the newly added vertex will require a set of data items present in the older vertex in order to satisfy the transaction instances it represents.

Recall that the weight of an edge corresponds to how many transaction instances need to access data on both vertices. Consider now the computation of an edge between vertices V_i

$$Flatten[Table[intersection,\ range\,of\,vars\,in\,vertex]$$

$$Flatten[Table[0 <= warehouseid1 <= 1, \{warehouseid, 0, 3\}]]$$

Figure 3.12: Structure and example of Mathematica query for edge weight computation

and V_j. Assume V_i is a vertex already present in the graph while V_j is currently being added. Further, assume there is an overlap between V_i and V_j . The weight of the edge between the two vertices will correspond to how many transaction instances in V_j need to access V_i to execute. Note that an access is formed by two components, a ρ formula and an input. If the ρ formula that causes the remote access is probabilistic, i.e. has a probability associated with it, then the weight of the edge is also multiplied by said probability. To compute this weight we leverage Wolfram Mathematica once more. We resort to a sequence of queries that allow us to count how many inputs mapped to V_j cause it to access V_i. Figure 3.12 shows an example of a query. The first operand corresponds to the intersection as identified during formula comparison. The second operand corresponds to the range of the variables in that intersection, according to the ϕ expression associated to the new vertex being added to the graph. The output of this query is a list of *boolean* values. The length of this list identifies how many inputs cause a remote access. Lastly, in the presence of a probabilistic access, we multiply the length of the list by the probability of the access.

Additionally, there can be multiple overlaps, i.e. overlaps on different ρ formulas, in which case we will store multiple edges between the same two vertices. However, all of these edges will be converted into a single edge in a subsequent stage.

After every ρ formula, present in the vertex that is being added to the graph, has been compared to the formulas in all previously existing vertices the vertex is ready to be added to the graph. There will no longer be any overlapping data items with any other vertex on the graph, instead there will be edges that resulted from subtracting items in order to avoid said overlaps. If replication was chosen over table split during the splitting phase there might still be overlaps between vertices, however these overlaps are only allowed to exist for replicated tables.

Before proceeding to the next vertex we need to compute the vertex weight. At this stage, the number of data items in the vertex is final and as such we can compute the vertex weight. Recall that we defined the weight of a vertex as the number of data items stored within it. Once again, we will leverage Wolfram Mathematica to perform this computation. We will first expand all the symbolic formulas in each vertex to compute which data items are accessed within each table and then sum how many data items are accessed in total across all tables. Note that if we

simply summed the number of data items accessed in each formula it would not be correct as there can be overlaps across the ρ formulas within a single vertex.

This process is repeated for all the vertices resulting from splitting. Upon completion we have the final graph with all the edges identified. On our implementation we also deploy a few optimizations to speed up this process, which we discuss next.

When adding a new vertex, containing a set of ρ formulas, we compare all of those formulas to the ones present in the vertices that have already been added to the graph. When an overlap is detected we subtract items from the new vertex, this consists of removing data items associated to a given ρ formula. Our first optimization consists of checking whether a ρ formula still references any data item after it undergoes the subtraction process. This allows us to avoid extra unnecessary comparisons by marking any ρ formula that no longer references any data item as remote.

Another optimization we implemented is the option of ignoring probabilistic ρ formulas whose probability is below a certain threshold during vertex comparison. This threshold is an optional parameter. While this approach provides us with a final graph that might contain overlapping vertices and, as a consequence, miss some edges, it also allows for much faster computation of the graph in scenarios where there are accesses that are very unlikely to happen and therefore should not be optimized for.

At this stage, we have a fully constructed graph that we will output to the next stage in our pipeline, the graph partitioning component.

3.5 Graph Partitioning

In this section we will detail how we prepare the graph we have built for the partitioning phase and specify which partitioning tool we use.

The graph built in the previous stage can have multiple edges between two vertices. This structure is not supported by the graph partitioner we are using and therefore we need to convert all of these edges into a single edge. This is a simple process in which we iterate over all the edges between two vertices and create one unique edge whose weight corresponds to the sum of all the individual edges between the vertices. After this simple process is completed, our graph is now ready to be partitioned.

We chose to use METIS as our partitioning tool as it is a well known state of the art graph partitioner. It provides a set of algorithms for graph partitioning, we selected *gpmetis*. It takes as input the graph in the form of a file, together with the number of partitions desired and, possibly, optional parameters.. The only optional parameter worth mentioning is *uFactor*. It

specifies the maximum allowed imbalance among the partitions. This parameter can be tuned to meet any requirements regarding the balance between partitions after partitioning. A solution in which every vertex goes within a single partition is not very interesting to analyze, this parameter avoids such scenarios. The number of partitions is also an important parameter as it specifies how many partitions the vertices will be distributed amongst. In our evaluation section we will analyze how the tuning of this parameter affects our database performance.

Note that balanced graph partitioning is a known NP-Complete problem. However, due to the work performed in the construction of our graph we end up with graphs with a limited amount of vertices, in the general case, which are partitionable by METIS in a matter of milliseconds.

The result given by METIS will provide us with the partition each vertex belongs to. After obtaining this we know which symbolic formulas belong to each partition and, as a consequence, which data items belong to which partition. This is our final partitioning scheme which is the output of our tool.

3.6 Discussion

The solution developed utilizes as much information extracted from symbolic execution as possible. By doing so it is able not only to formulate a graph partitioning problem but also avoid a one to one mapping between data items and partitions, this is due to the fact that we keep all formulas represented symbolically, allowing us to refer to a set of data items instead of individual items. This allows us to avoid any process that would result in a generalization of the mapping between data items and partitions with a loss of information. Lastly, we are able to obtain a partitioning scheme by solving the graph partitioning problem we formulated. Given our graph construction process and the work performed by our graph partitioner we are able to minimize the number of transactions that access multiple partitions hence maximizing database transaction throughput.

In the next section we will discuss our results and compare them to the ones obtained by other state of the art systems.

Chapter 4

Results

4.1 Evaluation

In this section, we will discuss the results obtained from experimental analysis of our system. We evaluate the quality of the partitions provided by Evolve to check whether or not we can improve the throughput of database systems.

As an evaluation metric, we will first use the percentage of distributed transactions in the systems workload. This metric shows how many transactions will be taking an execution time penalty due to the need of accessing data across multiple partitions. Note that an ideal partitioning scheme, which is not always possible due to the nature of the data, would result in zero distributed transactions. The second metric is the balance between the partitions obtained. A trivial partitioning solution that ensures zero distributed transactions would be to place every single data item in one partition. However, this is not scalable and we want to provide a solution that utilizes all available partitions and balances the data across said partitions. The balance factor between partitions is computed as the size of the largest partition divided by the size of the smallest partition.

We will use the TPC-C benchmark as our workload. Within the benchmark we will only consider update transactions and will ignore read only transactions. We are fairly more interested in analyzing the cost of distributed update transactions as they impose a higher cost during the execution of distributed transaction protocols. As a result, our workload is composed of 45% new order transactions, 45% payment and 10% delivery. For the new order transaction we have disabled the option for remote warehouses.

As a baseline, we will use our own implementation of the approach introduced by Schism[CJZM10]. This implementation does not comprise the machine learning phase. We are only interested in building a graph given an execution trace and then in partitioning the resulting graph. The

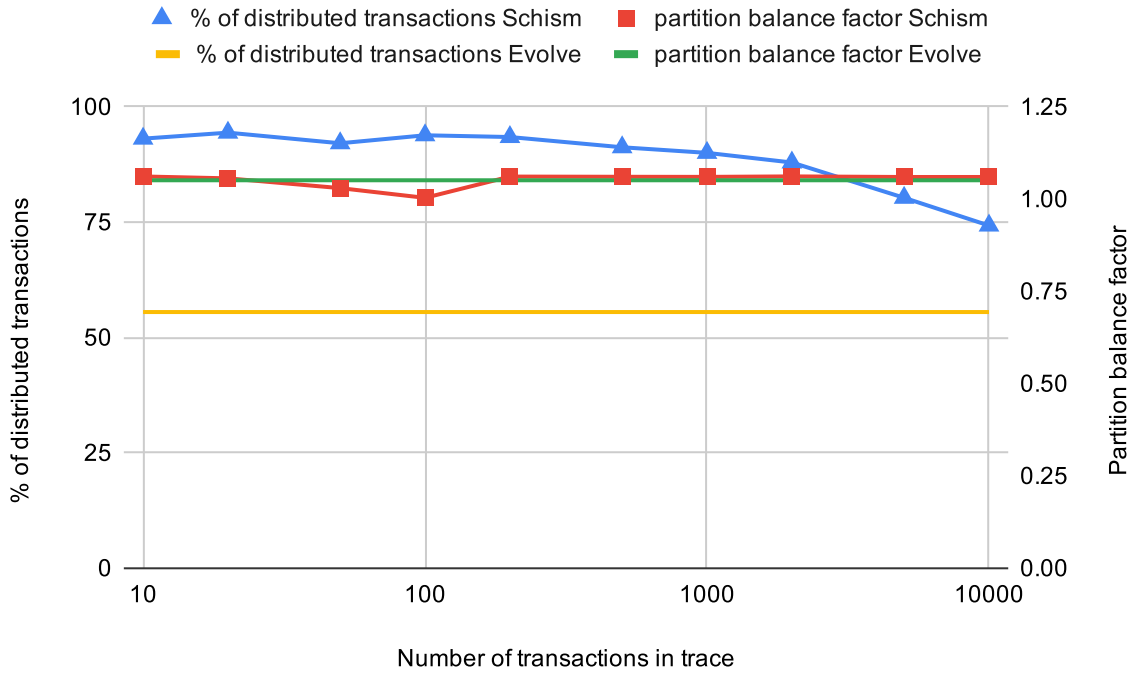

Figure 4.1: Analysis for one warehouse workload

partitioning scheme outputted by our implementation of Schism will then give us a baseline scheme to compare Evolve to.

It is also important to note that for the purpose of our experiments we ran the version of our system that adopts replication to deal with tables that can not be split by a splitting variable. This choice allows us to compare our approach to Schism as they also deploy replication in their system.

Figure 4.1 presents the percentage of distributed transactions and partition balance factor obtained by Schism and Evolve. For this experiment we used a TPC-C workload containing only one warehouse.

The results obtained show that as the number of transactions in the trace grows, Schism starts to perform better in terms of percentage of distributed transactions. This happens due to the fact that Schism relies on execution traces to identify the access patterns within the database. In the presence of a new transaction instance Schism places it in a random partition until a new partitioning scheme is calculated. This approach results in very bad results for small execution traces. In regard to the balance factor between partitions, Schism achieves a good result as the partitions generated have a balance factor close to 1.00 which is the optimal value.

Evolve does not depends on the number of transactions in a trace to perform partitioning. The partitioning scheme relies on symbolic execution data which is used to generate the graph.

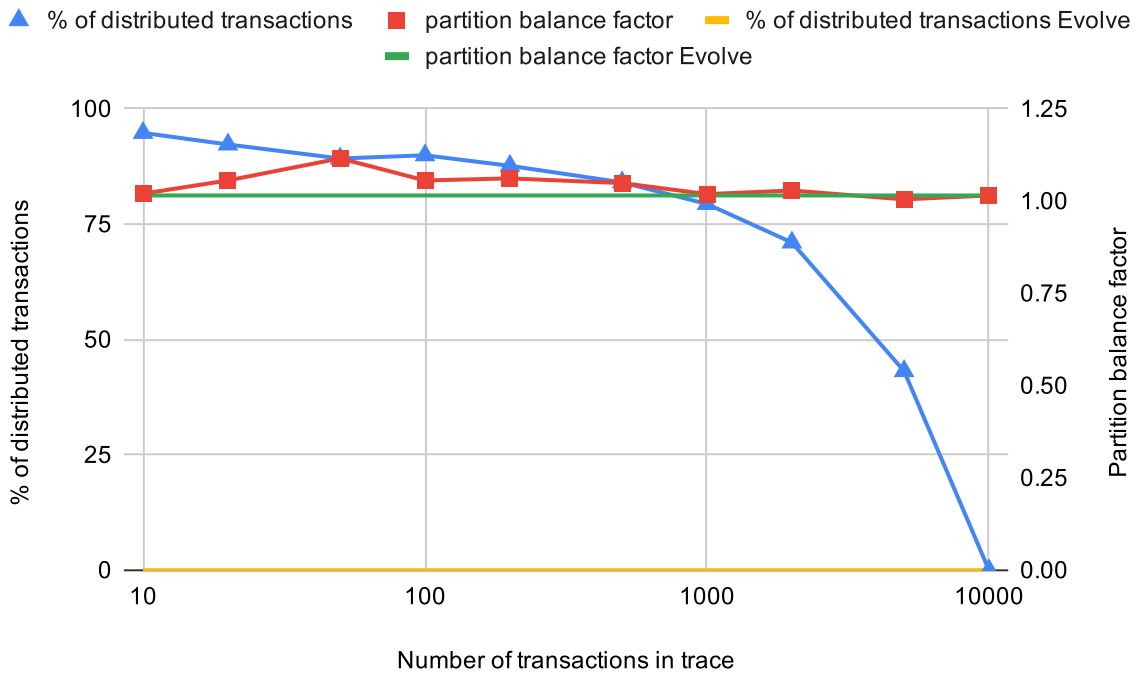

Figure 4.2: Analysis for two warehouses workload

As a result, Evolve is able to obtain better results in regard to the percentage of distributed transactions for this workload. Just like Schism, Evolve is also capable of balancing the partitions as the balance factor obtained also nears 1.00.

Figure 4.2 illustrates the results obtained for a workload with two warehouses using the same evaluation metrics as the first experiment.

This experiment shows the limitation of Schism once more. Without a sizeable execution trace Schism is not capable of achieving the same quality of results as Evolve. Additionally, this experiment shows that both Schism and Evolve are capable of partitioning the workload while minimizing the number of distributed transactions. The difference is that Evolve does not rely on the size of the execution trace and Schism requires an execution trace containing 10000 transactions to be able to partition optimally. The balance factor for both systems shows that both of them are capable of generating balanced partitions.

The results show that both systems are capable of providing partitioning schemes that do not include any distributed transactions for a workload with two warehouses. This results in a very high system performance as there is no delay imposed by distributed transactions.

It is interesting to analyze what happens for a workload with one warehouse. The best partitioning for TPC-C is known to be by warehouse. By testing the system with a workload containing only one warehouse that partitioning strategy is no longer possible. The results in

Figure 4.1 show that our system performs better than Schism in this scenario. This is a very encouraging result as it proves Evolve can perform in an adversarial scenario.

Our results prove that our approach is competitive in comparison to Schism. We are able to generate partitions that do not contain any distributed transactions. The balance between partitions is also satisfactory as partitions have very similar sizes in our experiments. Lastly, our tool is capable of achieving the same partitioning result as Schism, this result is the best known partitioning for TPC-C and has been proved by human experts[SMA$^+$07]. By doing so without requiring execution traces and, instead, using symbolic execution we reach the goal proposed for this work.

Chapter 5

Conclusions

The overwhelming amount of data stored in databases does not allow for fully replicated systems to scale. It is necessary to distribute the load across multiple machines while providing systems that do not impose unreasonable delays.

In this report we present a new approach to data partitioning, a strategy that leverages symbolic execution for multiple purposes and by doing so it seeks to improve database performance. Symbolic analysis provides us with fine-grained knowledge on the queries made to the system. Furthermore, this knowledge is used to formulate a graph partitioning problem. Lastly, by solving the formulated problem we obtain a partitioning scheme.

Symbolic execution has never been exploited with this purpose and as such we believe the system we propose is a singular case in the literature. Our results show that Evolve can produce balanced partitions that minimize the number of cross partition transactions and thus improves database transaction throughput.

Bibliography

[ARK06] Amine Abou-Rjeili and George Karypis. Multilevel algorithms for partitioning power-law graphs. 2006.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. Rwset: Attacking path explosion in constraint-based test generation. 2008.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. 2008.

[CJZM10] Carlo Curino, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. Schism: a workload-driven approach to database replication and partitioning. 2010.

[CRS+08] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. 2008.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. 2007.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 1978.

[LFEK+19] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. Dynastar: Optimized dynamic partitioning for scalable state machine replication. IEEE, 2019.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. 2010.

[PRRR13] Joao Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. {AUTOPLACER}: Scalable self-tuning data placement in distributed key-value stores. 2013.

[Sch90] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. 1990.

[SMA+07] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). 2007.

[Vie18] Miguel Cândido Viegas. Fine grained transaction scheduling in replicated databases via symbolic execution. 2018.

[VRCG+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot: A java bytecode optimization framework. 2010.

[Wol] Wolfram Research, Inc. Mathematica 8.0.