

# Database Partitioning via Symbolic Execution

Francisco Jorge Silva Santos Rola  
Instituto Superior Técnico, Universidade de Lisboa

**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.

## I. INTRODUCTION

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.

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 [1] 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 [2]. 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 [1] or building complex graphs [2] 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.

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.

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 report

## II. RELATED WORK

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 about the state of the art in data partitioning. Background on symbolic execution will be provided in section 3 where we present our system.

### A. Data Partitioning

This section explores recent work in the data partitioning field, also known as data sharding. Schism [1] 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 [2] is a recent work that also leverages a graph partitioner but builds it on the fly instead and thus works online. Lastly, Autoplacer [3] 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.

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. First, 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. Second, 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.

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.

System	Mode	Problem	Workload info	Opt.
Schism	Offline	Graph Part	Execution Traces	ML
Dynastar	Online	Graph Part	Workload analysis	None
Autoplacer	Online	ILP	Workload analysis	ML
Evolve	Offline	Graph Part	Symbolic Execution	SE

TABLE I: Summary of data partitioning solutions

2) *Dynastar*: 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.

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.

3) *Autoplacer*: 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. It formulates the data placement problem as Integer Linear Programming problem and applies machine learning techniques to generalize its findings.

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.

### B. Discussion

Table I 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. 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). 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. 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 (ML).

After discussion, it is clear that these systems have disadvantages. In the following section, we present Evolve that seeks to leverage symbolic execution to solve the data parti-

tioning problem whilst avoiding any of the downfalls identified in other systems.

### III. SYSTEM/APPROACH

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.

First, the symbolic analysis stage is responsible for analyzing the code of each transaction<sup>1</sup> 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. 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.

#### A. Symbolic Analysis

State of the art database partitioning solutions such as Schism [1] and Dynastar [2] 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

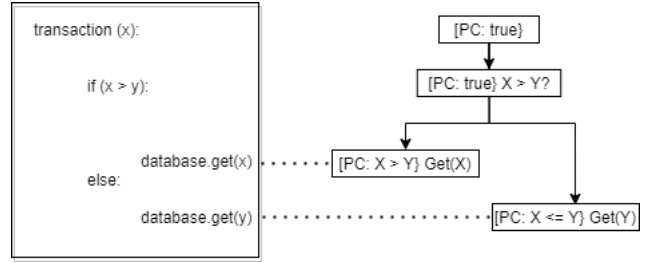


Fig. 1: Symbolic execution example. PC stands for Path Condition

a transaction is executed, one of those paths will be followed accessing a set of data items. 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 1 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 1 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

<sup>1</sup>this can be executable or the source code

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 [4] 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. 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.

## B. 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  $\rho$  while the range of its input variables is defined as  $\phi$ . Each  $\rho$  always has an associated  $\phi$ .

Moreover, it is possible to provide a symbolic formula that contains two different  $\rho$  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  $\rho$  with an associated probability as a probabilistic  $\rho$ .

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

The initialization process is as follows. Each transaction profile identified by the symbolic analysis is parsed through. This profile will contain a set of  $\rho$  and associated  $\phi$ . The first step is to analyze each  $\rho$  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.

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.

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  $\rho$  formulas containing indirect reads we can generate the  $\sigma$  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

### C. 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.

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 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

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

Fig. 2: Symbolic formulas for the payment transaction in TPC-C

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.

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  $\sigma$  containing a set of  $\rho$  formulas and respective  $\phi$ . These formulas will span a multitude of tables depending on the transaction profile. Figure 2 presents the formulas obtained for the payment transaction profile in the TPC-C benchmark. In this example we can see four different  $\rho$  formulas. There are also four different tables being accessed identified by the prefixes in each of the formulas.

The first step is to group  $\rho$  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  $\rho$  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 2, if we chose to perform this split we would have four disjoint sub-vertices, each containing a single  $\rho$  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.

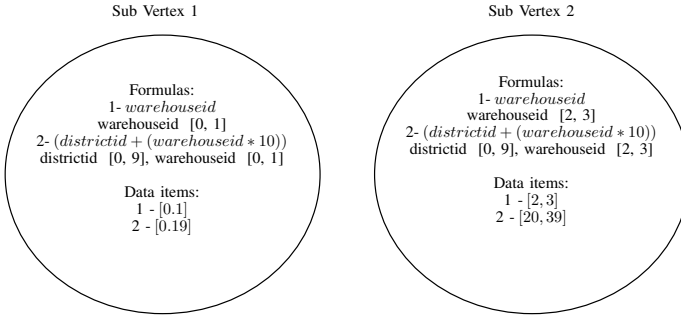


Fig. 3: Vertices resulting from a split by *warehouseid*

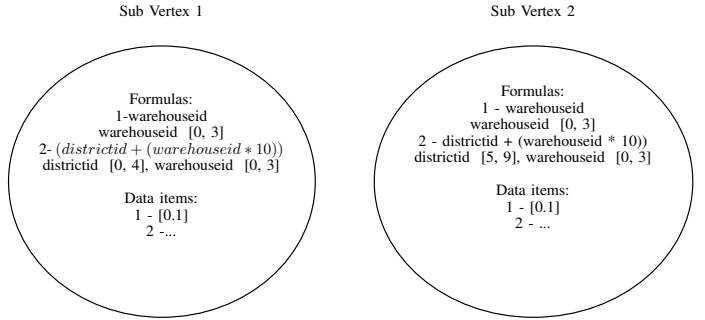


Fig. 4: Vertices resulting from a split by *districtid*

Another possible approach is to reason on splits based on partitioning the input range of the symbolic variables in the  $\rho$  formulas. Recall that each  $\rho$  always has an associated  $\phi$ , 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  $\phi$  expressions.

For the example in figure 2, consider a split in two distinct sub-vertices by splitting the original vertex based on the range of the *warehouseid* variable. Let us assume  $\text{warehouseid} \in [0, 3]$  originally. A possible split would be to assign  $\text{warehouseid} \in [0, 1]$  to one sub-vertex and  $\text{warehouseid} \in [2, 3]$  to the other sub-vertex. Figure 3 shows this split, for simplicity we will 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  $\rho$  formulas and respective  $\phi$  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 4 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 *districtid* 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  $\rho$  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  $\rho$  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  $\rho$  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  $\rho$  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. We implemented an algorithm that collects 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.

Initially, we iterate over all the  $\rho$  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  $\rho$  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  $\rho$  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 [5]. We build a FindInstance query which will have as operands both  $\rho$  formulas under analysis and their respective  $\phi$ .

At this point we come across another problem. If there are more than two  $\rho$  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.

If the formulas have a variable in common. 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. This allows us to add the common splitter identified 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 disjointness, this will only be done in a subsequent step. Following the replication approach at this stage would simply marking certain  $\rho$  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  $\rho$  formulas via manipulating its associated

$\phi$ , as we do for the regular splits, we constrain the value that each  $\rho$  formula can yield directly.

This approach allows us to group together any overlapping data items between  $\rho$  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  $\rho$  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  $\rho$  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.

After this phase we are done collecting the candidate splitters and are ready to proceed.

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  $\rho$  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  $\rho$  formulas that belong to any other table will be under analysis.

Initially, we build an array with as many entries as  $\rho$  formulas across the tables that need 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  $\rho$  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 4. 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  $\rho$  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  $\rho$  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 achieves full coverage then any variable added to the set of splitting variables will increase the edge weight as it will not be able to cover, at least, one  $\rho$  formula. This is what happened in the TPC-C example in Figure 4 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 splitting 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  $\rho$  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  $\rho$  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.

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  $\rho$  formulas uniformly, by manipulating the  $\phi$  associated with each  $\rho$ .

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.

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.

#### D. 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.

For this purpose we implemented an algorithm that 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 comparisons, 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  $\rho$  formula and associated  $\phi$  in the first vertex with all of the  $\rho$  formulas and associated  $\phi$  on the second vertex. As an optimization, it is only necessary to compare  $\rho$  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.

With the objective of detecting overlaps between  $\rho$  formulas across different vertices we deploy Wolfram Mathematica once more. We use *Reduce* queries for this purpose.

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  $\rho$  formula. This allows us to concatenate the result of the Mathematica queries into our  $\rho$  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  $\rho$  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  $V_i$  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  $\rho$  formula and an input. If the  $\rho$  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  $V_i$ . 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  $\rho$  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  $\rho$  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  $\rho$  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  $\rho$  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  $\rho$  formula. Our first optimization consists of checking whether a  $\rho$  formula still references any data item after it undergoes the subtraction process. This allows us to avoid extra unnecessary comparisons by marking any  $\rho$  formula that no longer references any data item as remote.

Another optimization we implemented is the option of ignoring probabilistic  $\rho$  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.

### E. 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 *gpmets*. 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.

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

#### IV. 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 [1]. 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 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 5 presents the percentage of distributed transactions and partition balance factor obtained by Schism and Evolve.

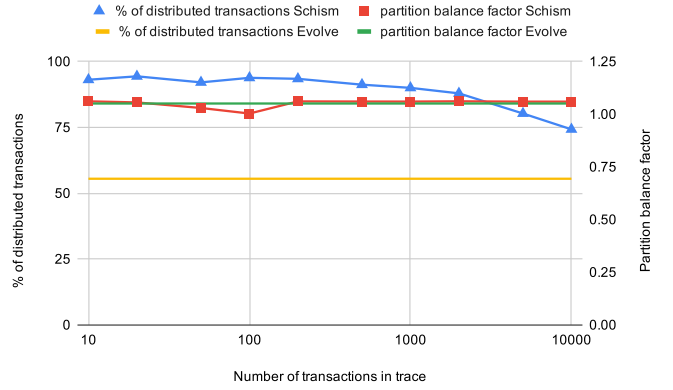


Fig. 5: Analysis for one warehouse workload

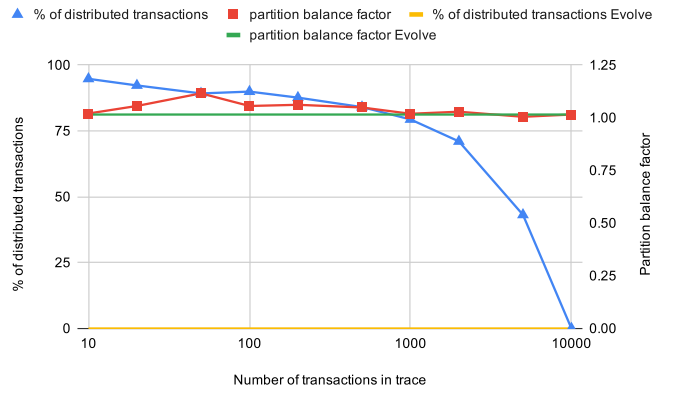


Fig. 6: Analysis for two warehouses workload

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 depend 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. 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 6 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 5 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 [6]. By doing so without requiring execution traces and, instead, using symbolic execution we reach the goal proposed for this work.

## V. CONCLUSION

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.

## REFERENCES

[1] C. Curino, E. P. C. Jones, Y. Zhang, and S. R. Madden, "Schism: a workload-driven approach to database replication and partitioning," 2010.

[2] L. H. Le, E. Fynn, M. Eslahi-Kelorzai, R. Soulé, and F. Pedone, "Dynastar: Optimized dynamic partitioning for scalable state machine replication." IEEE, 2019.

[3] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues, "{AUTOPLACER}: Scalable self-tuning data placement in distributed key-value stores," 2013.

[4] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," 2010.

[5] Wolfram Research, Inc., "Mathematica 8.0." [Online]. Available: <https://www.wolfram.com>

[6] 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.