Automated Data Partitioning for Highly Scalable and Strongly Consistent Transactions

Alexandru Turcu
Virginia Tech
talex@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

ABSTRACT

Modern transactional processing systems need to be fast and scalable, but this means many such systems settled for weak consistency models. It is however possible to achieve all of strong consistency, high scalability and high performance, by using fine-grained partitions and light-weight concurrency control that avoids superfluous synchronization and other overheads such as lock management. Independent transactions are one such mechanism, that rely on good partitions and appropriately defined transactions. On the downside, it is not usually straightforward to determine optimal partitioning schemes, especially when dealing with non-trivial amounts of data. Our work attempts to solve this problem by automating the partitioning process, choosing the correct transactional primitive, and routing transactions appropriately.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation*; H.2.4 [**Database Management**]: Systems—*Distributed databases*

General Terms

Performance

Keywords

Data Partitioning, Transactional Store, Distributed Transactional Memory, Independent Transactions

1. INTRODUCTION

Distributed transactional storage systems nowadays require increasing isolation levels, scalable performance, fault-tolerance and a simple programming model for being easily integrated with transactional applications. The recent growth of large scale infrastructures with dozens or hundreds of computational nodes needs transactional support ready to scale with them.

Many of the modern transactional storage systems have abandoned strong consistency (e.g., serializability) in order to achieve good scalability and high performance [4, 10, 17]. Weak consistency models (e.g., eventual consistency) incur the expense of allowing some non-serializable executions, which, if at all tolerated by the application requirements, are more difficult to deal with for the developers [28]. In fact, it was observed that developers prefer strong consistency when possible [6].

For this reason, transactional storage systems that offer serializability without forsaking high speed and high scalability, represent a very promising sweet spot in the overall design space. One system that approaches this sweet spot for On-Line Transaction Processing (OLTP) workloads is Granola, as proposed by Cowling and Liskov in [8]. Granola employs a novel transactional model, *independent transactions*, to keep overheads and synchronization to a minimum while guaranteeing serializable distributed transactions. To help reach its high transaction throughput, Granola relies on storing the data in main memory and operating upon it using transactions expressed in the application's native programming language, as opposed to a query language like SQL. This essentially qualifies Granola as a Distributed Transactional Memory (DTM) system.

One key enabler for good performance in the Granola model is having the data organized in fine-grained, high-quality partitions that promote the use of single-partition and independent distributed transactions. This can be considered a drawback for Granola, as developers need to manually organize the data, choose the transaction primitives, and route transactions appropriately. This work focuses on eliminating this drawback by automating the three tasks.

To reach our goal, we adapt and extend an existing graph-based data partitioning algorithm, Schism [9], originally proposed for traditional, SQL-based databases. Our major contributions are:

- We extend Schism to support independent transactions. The extended algorithm will propose partitions that favor fast single-partition and independent transactions against the slower, Two-Phase Commit (2PC) coordinated transactions.
- We develop a mechanism based on static program analysis for determining edge weights in the graph that Schism uses for proposing partitions. This essentially enables applying an algorithm like Schism to independent transactions, or, more generally, to any DTM-style transactions expressed in a native programming language.

- We develop a routing mechanism based on machine learning, for routing transactions to an appropriate set of partitions. This is essential for enabling any kind of automatic partitioning for Granola or any other DTM-style environment, where a transaction's access set is not known a priori.

Additional minor contributions include automatic program refactoring for run-time transaction trace collection, and automatic choice of an appropriate transaction primitive based on static program analysis. To the best of our knowledge, this is the first work that provides an end-to-end automated framework for exploiting independent transactions.

We frame our work in a DTM environment for two reasons: (i) similar environments were shown to support much higher transaction throughputs than traditional Relational Database Management Systems (RDBMS) for OTLP workloads [29], and (ii) our choice presents us with some very interesting problems that allow us to innovate.

The rest of the paper is organized as follow. In Section 2 we describe the Granola model and the Schism technique. Section 3 overviews the system model. The automatic framework is presented in Sections 4, 5, 6. In Section 7 the framework's evaluation is reported. Section 8 discusses past and related works and Section 9 concludes the paper.

2. BACKGROUND

2.1 Granola: Independent Transactions

Granola [8] is a transaction coordination infrastructure proposed by Cowling and Liskov. Granola targets On-Line Transaction Processing (OLTP) workloads. Granola is a Transactional Memory (TM) system, as it expresses transactions in a native programming language and operates on data stored in main memory for performance reasons. Synchronization overheads are kept to a minimum by executing all transactions within the context of a single thread. This approach reduces the need for locking, and was shown to significantly improve performance compared to conventional databases in typical OLTP workloads [29, 12, 15].

Granola employs a novel timestamp-based transaction coordination mechanism that supports three classes of one-round transactions. *Single-Repository Transactions* are invoked and execute to completion on one repository (partition) without requiring any network stalls. *Coordinated Distributed Transactions* are the traditional distributed transactions that use locking and perform a two-phase commitment process. Additionally, Granola proposes *Independent Distributed Transactions*, which enable atomic commitment across a set of transaction participants, without requiring agreement, locking and with only minimal communication in the coordination protocol.

Single-repository and independent transactions execute in *timestamp mode*. These transactions are assigned an upcoming timestamp, and execute locally in timestamp order. Repositories participating in an independent distributed transactions need to coordinate to select the same timestamp. Each participant proposes a timestamp for executing the transaction, and broadcasts its proposal (vote) to the other participants. Once all votes are received, the final timestamp is selected locally as the maximum among all proposals. This selection is deterministic, and the coordination it requires is very light-weight (needs only one messaging round). At the selected time, the transaction can execute without any stalls or network communication.

In order to execute coordinated transactions, the repository needs to switch to *locking mode*. In locking mode, all transactions must acquire locks (thus incurring overheads), and can not use the fast timestamp-based execution. Furthermore, coordinated transactions must undergo a slow two-phase commit. The repository can revert to timestamp mode when all coordinated transactions have completed.

Granola provides strong consistency (serializability) and fault-tolerance. Data is partitioned between the Granola repositories – with each repository managing one partition – although it is also possible to keep some of the data replicated between repositories to improve performance. Each repository consists of one master and several replicas. The replicas are used for fault-tolerance, not for scalability. Most transactions must be executed by the master node of each repository – the only exception is for read-only, single-repository transactions, which can be serviced by replicas.

In Granola, single-repository and independent distributed transactions never conflict, because they are executed sequentially using a single thread. This means mechanisms employed for rollback and aborts, such as locking and undo- or redo-logging, are not needed for these transaction classes, reducing overheads and improving performance.

Granola transactions do have restrictions that limit their applicability and place further requirements on the potential partitioning schemes:

- Independent transactions must reach the same commit decision, independently, on every participating repository. This is possible when the transaction never aborts (e.g., read-only transactions), or the commit decision is based on data replicated at every participating repository.
- All transactions must be able to complete using only data available at the current repository. This is a firm requirement for single-repository and independent transactions, but could potentially be relaxed for coordinated transactions.

Performance in Granola depends on how the workload and partitioning scheme are able to exploit fast single-repository and independent transactions. The user must manually define the partitioning scheme, implement the transactions using the appropriate classes, and route transactions correctly. Furthermore, the partitioning scheme must be compatible with the Granola restrictions outlined above. This paper aims to automate this partitioning process.

2.2 Schism: Graph-Based Partitioning

Curino et al. presented Schism [9], the approach for automated data partitioning that we build upon in our present work. Besides lacking support for independent transactions, Schism as it stands can not be applied to stored-procedure style DTM transactions, which further motivates our work. For the sake of completeness, in this section we overview Schism and describe how it works.

Schism takes as input a representative workload in the form of an SQL trace, and the desired number of partitions. It then proposes partitioning and replication schemes that minimize the proportion of distributed transactions, while promoting single-partition transactions. This is done in order to increase performance, as single-partition transactions are fast. The proportion of distributed transaction repre-

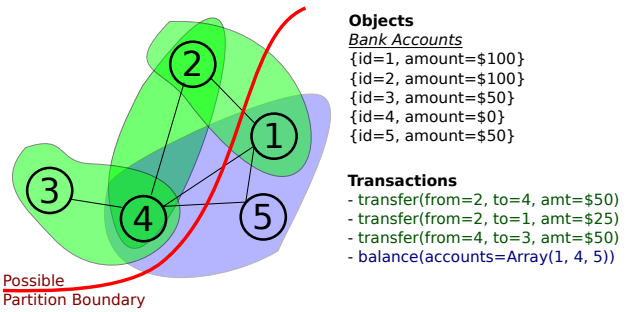


Figure 1: Example graph representation in Schism. The shaded areas are the transactions, which are represented in the graph by edges connecting all accessed objects.

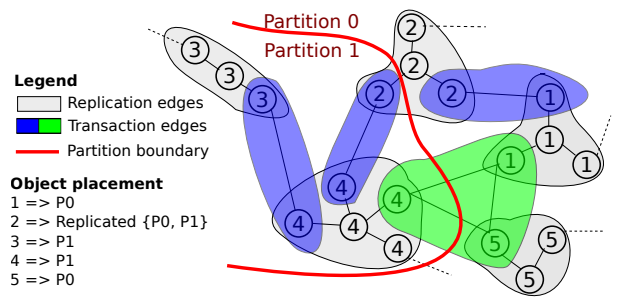


Figure 2: Example graph representation in Schism, with replication.

sents a measure of the *partitioning quality*. The fewer distributed transactions there are, the higher the quality of the partitioning. The partitioning process has four phases:

First, the **graph representation** phase converts the SQL trace into a graph. Nodes in this graph represent data items (database tuples/transactional objects) that were encountered in the trace. Two nodes are connected by an edge if they were accessed together within the same transaction. Thus, the representation of a transaction takes the form of a clique: the tuples accessed by the transaction are all interconnected. An example is shown in Figure 1. A number of heuristics are applied to promote scalability, such as tuple and transaction sampling, and coalescing tuples accessed by the same set of transactions into a single node.

The graph is then modified by replacing each node with a star-shaped configuration of nodes. This is done in support for data replication. A node A which previously had n neighbors, is replaced by $n+1$ nodes: one in the center, A_0, which is connected to n new nodes $(A_1 ... A_n)$ by edges representing the cost of replicating the original node A. Each of these new nodes is then connected by a single edge to another node representing the original neighbors. This processing can also be explained as replacing each edge in the original graph by three edges connected in sequence: the two outer edges represent the cost of replicating the data, and the middle edge represents the cost of entering a distributed transaction. An example is illustrated in Figure 2.

In the **partitioning phase**, the previously constructed graph is partitioned using a standard k-way graph partitioning algorithm. The authors used the program METIS [16] for this purpose. This is an optimization problem, where the primary target is minimizing the cumulative cost of the edges that cut across partitions. This is equivalent to minimizing the number of distributed transactions. A secondary target is balancing the partitions with respect to node weights. Node weights can be assigned based on either data size, or number of transactions, depending on whether the partitions should be balanced in storage size or load.

For small workloads, the output of the partitioning phase can be used as-is, by means of a lookup table. Newly created tuples would initially be placed on a random partition, while a separate background task periodically recomputes the lookup table and migrates data appropriately. This method however can not be applied to large datasets for two reasons: (i) creating and partitioning the graph without sampling is limited by the available memory and processing time, and (ii) the lookup table size is similarly limited by the available memory.

These reasons motivated Schism's **explanation phase**. In the explanation phase a more compact model is formulated to capture the tuple $\rightarrow$ partition mappings as they were produced in the partitioning phase. Schism does this by employing machine learning, or more specifically, C4.5 decision trees [25] as implemented in the Weka data mining software [19]. The resulting models are essentially sets of range rules, and are useful if they satisfy several criteria: they are based on attributes present as `WHERE` clauses in most SQL queries, they do not significantly reduce the quality of the partitions by misclassification, and finally, they work for new, previously unseen queries, as opposed to being overfitted to the training set. To satisfy these criteria, the authors employed strategies such as placing limitations on the input attributes to the classifier, using aggressive pruning and cross-validation, and discarding classifiers that degrade the partitioning quality.

Lastly, the final partitioning scheme is chosen in the **final validation** phase. The candidates considered are (i) the machine-learning based range rules, (ii) the fine-grained lookup table, (iii) a simple hash-based partitioning, and (iv) full-table replication. The scheme chosen is the one with the fewest distributed transactions. In case two schemes lead to similar results, the simpler of the two is chosen.

3. SYSTEM OVERVIEW

Our partitioning methodology was designed and implemented in the context of a Distributed Transactionl Memory (DTM) system. DTM systems store data in main-memory, and access it using transactions expressed in a programming language (usually the same as the rest of the application), as opposed to a separate query language. Declaring and running transactions in DTM should be as simple as possible: ideally the transaction code is simply written inside an *atomic block*, as exemplified in Figure 3(a).

Our choice of environment (DTM) and transaction model (Granola's independent transactions) make Schism impossible to apply directly, for several reasons:

- Schism does not support independent transactions. Any distributed transactions in Schism would have to be 2PC-coordinated, which degrades performance.
- Schism makes no effort to prevent data dependencies across partitions. At best, such dependencies are incompatible with independent transactions. At worst, they are incompatible with Granola's single-round transaction model, leading to unusable partitions.

```
atomic {

    acc1.amt += value
    acc2.amt -= value
}
```
a

```
atomic { implicit txn =>
    val acc1 = Hyflow.dir.open[BankAccount]("acc1")
    val acc2 = Hyflow.dir.open[BankAccount]("acc2")
    acc1.amt() += value
    acc2.amt() -= value
}
```
b

Figure 3: Example atomic blocks. In *a.* objects are assumed to not need opening before being accessed, as is common for Software Transactional Memory (STM). *b.* shows the same atomic block written to Hyflow's API, also including object opening.

- Schism assumes transactions are expressed in SQL code, whose WHERE clauses can trivially be inspected to obtain information about the dataset of a transaction, which is then used to route each transaction to the appropriate partitions. Given that transactions in our system are not expressed in parsable query code, but are stored procedures written in a programming language, the task of routing transactions becomes significantly more complicated.

Our implementation is based around Hyflow2 [32], a JVM-based DTM framework written in Scala. We implemented the Granola protocol in this DTM framework. Unlike Granola, which relies on opaque up-calls from the framework to the application and lets the application code handle locking and rollback mechanisms when needed, we opted to provide a more friendly API and let the framework deal with these mechanisms. Figure 3 (b) shows an example transaction.

3.1 Partitioning Process

This section provides a brief description of the partitioning process. In a production system, this process would run periodically alongside transaction processing, and dynamically migrate objects at run-time. Our implementation however, being only a prototype, performs the partitioning off-line.

The first phase in our partitioning workflow performs **static analysis and byte-code rewriting** on all transactional routines in the workload. This step serves three purposes. Firstly, it collects data dependency information which is later used to ensure the proposed partitioning schemes are able to comply to our chosen one-round transactional model (no data dependencies are allowed across partitions). Secondly, it extracts summary information about what operations may be performed inside each atomic block, to determine whether an atomic block is abort-free or read-only. Finally, each transactional operation is tagged with a unique identifier to help make associations between the static data dependencies and the actual objects accessed at run-time.

The second phase is **collecting a representative trace** for the current workload, which includes a record for every transactional operation performed. Each record contains the transaction identifier, the type of operation, the affected object, and the operation's identifier as previously tagged.

The next three phases are similar to the corresponding phases in Schism. The **graph representation phase** converts the workload trace into a graph where nodes represent objects and edges represent transactions. This graph is governed by the same rules as in Schism (see Section 2.2). Additionally, edge weights are updated to reflect the new transaction models, along with their restrictions and desirability. The graph is then partitioned using METIS in the **partitioning phase**. The result from this step is a fine-grained association from object identifiers to partitions. A concise model of these associations is created using WEKA classifiers in the **explanation phase**.

The final phase is concerned with **transaction routing and model selection**. While in Schism routing information was easily extracted from the `WHERE` clause of SQL queries when available, our atomic block model for expressing DTM transactions prohibits using a similar approach. We thus introduce a machine-learning based routing phase. The data used to train this classifier is derived from the workload trace, using the object-to-partition mapping. Finally a transaction model is selected for every transaction class based on the number of partitions it needs, whether it may abort, and whether it writes any data (or is read-only).

3.2 Run-Time Behavior

During the previously described process, we train two sets of classifiers. The first set is tasked with object-to-partition mapping. These classifiers determine the object placement, and we will call them the *placement classifiers*. While it may reduce the quality of the resulting partitions, misclassification at this stage is mostly harmless, since it is the classifier that dictates the final object placement.

The second set of classifiers are the *routing classifiers*. They are used on the client side (i.e., in the thread that invokes the transaction) to decide which nodes to contact for the purpose of executing the current transaction. Due to the transactions being expressed as regular executable code, this information is not readily available until the code is run. Inputs for these classifiers are the parameters passed to the transaction. Misclassification at this stage has the potential to be harmful, as a misrouted transaction may not have access to all objects needed to execute successfully. We address this situation by allowing such a misrouted transaction to abort and restart on a larger set of nodes.

Finally, we do not require users to be aware of the partitioning scheme or the transaction execution model when writing transaction code. Thus, users should be able to write a single atomic block, and the system would make sure the appropriate code branches will execute at the corresponding partitions. In our prototype implementation, the same code is expected to execute properly on all partitions. This requires a defensive programming style, which checks that the return value of certain object open operations is not *null*. While this is a good practice anyway for error handling, our current implementation explicitly uses *null* references to denote an object is located at another partition.

4. STATIC ANALYSIS

Our static analysis phase is motivated by three factors: (i) determining data dependencies in order to avoid dependencies across partitions, (ii) determining which transactions can abort in order to choose the correct transaction model, and (iii) help with recording workload traces. Simply observing runtime behavior is insufficient — for instance, observing a particular transaction profile never aborted as recorded in a runtime trace does not constitute a guarantee that it can never abort.

Our static analysis phase is implemented using the Soot Java Optimization Framework [18]. Since we operate on JVM bytecode, few of the mechanisms described in this section are actually specific to Scala — transactions could just as easily be expressed in Java, with only simple changes re-

quired to the static analysis mechanisms. We make several passes over every application method.

4.1 First Pass

The first pass serves three purposes: (i) it identifies transactional methods, (ii) it tags transactional operations, and (iii) it records associations between the classes Scala uses for anonymous functions and their main method which contains the actual application code.

To identify transactional methods, we iterate over all units of each method (units are Soot's abstraction over the JVM byte-code). We look for invocations of certain methods and references to objects of certain classes that are usually associated with transactions — these are listed in Table 1. Methods that match are recorded as *transactional methods*.

In addition to recording transactional methods, we tag units representing invocations to the methods in Table 1. Tags are a feature in Soot that can associate information with any unit, for easier retrieval. Within the tag we store what kind of transactional operation this invocation represents (e.g., object open, object delete, field read, field write, transaction abort, etc.), and an integer uniquely identifying each invocation site (we name this integer the *tag id*).

Scala uses classes inheriting *AbstractFunctionN*[1] to implement anonymous functions (closures). The application code is usually located in a method named *apply* which takes arguments of the appropriate types. Scala however defines another polymorphic method with the same name, but with arguments of type Object (the root base class on the JVM). This method acts as a stub — its purpose is to convert (typecast or un-box) all arguments to the correct specific type and call the *apply* method containing the application code. For the purpose of our static analysis the stub method is not interesting. We thus record the association between the *AbstractFunctionN*-derived class and the *apply* method containing application code, but only if *apply* is a transactional method as defined above.

4.2 Second Pass

Once all transactional methods and transactional anonymous function classes are known, we construct a static invocation graph. This is done in the second analysis pass. As before, we pay attention to method invocations, but this time our targets are the previously identified transactional methods. We first add all transactional methods as nodes in the invocation graph. Any invocation of method g from within method f adds to the graph directed edge $f \rightarrow g$.

Besides direct invocations of transactional methods, we also add indirect invocations to the graph. Scala is a functional language and has support for higher-order functions (functions that take other functions as parameters). An invocation site is included in the graph when a previously identified transactional *AbstractFunctionN* object is passed to a higher-order function (either user-defined, or from the standard library: map, filter, etc.). The edge added to the static invocation graph points from the invoking function f to the *apply* method of the transactional *AbstractFunctionN* object, which is invoked indirectly by the higher-order function. Alongside constructing the static invocation graph, all invocation sites (direct and indirect) are tagged as before.

[1]Where N is an integer standing for the number of arguments taken by the function.

4.3 Third Pass

The third analysis pass extracts internal data dependency information for each transactional method. It processes each method, taking as input its bytecode as tagged in passes 1 and 2. The output is a directed graph representing data dependencies between the various accessed objects and external methods invoked. Firstly, nodes are created in the output graph for important transactional operations that are the target of the dependency analysis. Such operations are object open, create, delete, transaction abort, and also external method invocations, as tagged in previous steps.

This pass is implemented as a forward data-flow analysis. Each Soot unit has an associated state data-structure that can hold a representation of its dependencies. This representation has two parts: (i) a set of node dependencies and (ii) a set of value dependencies. A node dependency occurs when the result of an important operation (i.e. a transactional object that has been opened) is used in a subsequent statement. Value dependencies occur when any other (i.e., non-node) value is used in a subsequent statement. The latter do not have a presence in the dependency graph, but help propagate dependencies between nodes.

Initially, all the state data-structures are empty. We identify the direct dependencies for every Soot unit, and categorize them into two sets, for node and value dependencies. The value dependencies are traced back to the *origin unit* that defined each of the values. The states associated with the origin units are then retrieved and merged. We further merge this state with a state object formed from the value and node dependencies. Finally, we store the resulting state for the current Soot unit. Pseudocode for this process is shown in Algorithm 1.

Algorithm 1 Forward data-flow analysis pseudocode.

```
for each unit ← allUnits do
    allDeps ← unit.getDirectDeps()
    (nodeDeps, valueDeps) ← allDeps.partition( isNodeDep _ )
    valueDeps_originUnits ← valueDeps.getOriginUnits()
    valueDeps_states ←
          getStateForAll( valueDeps_originUnits )
    merged_valueDepState ← mergeAll( valueDeps_states )
    currentState ← new DepState( valueDeps, nodeDeps )
    newState ← merge( currentState, merged_valueDepState )
    storeStateForUnit( unit, newState )
end for
```

After the data-flow analysis, we construct the dependency graph. Starting with an empty graph, we add nodes for all the units of interest. Then we iterate over all nodes A in the graph, adding edges from from B to A, for all node-dependencies B of A.

We illustrate this process in Figure 4. The source code to be analyzed is shown in Figure 4(a). Notice how many intermediate values are held in variables of their own. This emulates the behavior of Soot, which will indeed use separate locations for every intermediate value, greatly simplifying the static analysis. For clarity, we show a simplified version. In Figure 4(b) we show the direct dependencies of each node and value in the code. Following the data-flow analysis, units have an associated state storing all their dependencies, shown in Figure 4(c) as the set of all edges pointing to a particular block. Finally, the dependency graph is created by discarding all non-node values (Figure 4(d)).

Method signature / *Ref type*	Description
TxnExecutor.apply (block: Function)	Invokes a transaction given an anonymous function as an atomic block. The block can potentially be a top-level transaction, and thus an entry point for the subsequent analysis phases.
TxnExecutor.apply (name: String, args: Array)	Invokes a pre-registered transaction given its name.
Hyflow.registerAtomic (name: String, block: Function)	Registers an atomic block to execute as a transaction when invoked by name.
HRef.apply() HRef.update(val: X)	Reads and writes, respectively, a field of a transactional object. Used to extract data dependencies between objects. Also used to identify read-only transactions.
Txn.rollback() Txn.retry()	Permanently or temporarily aborts a transaction. Used to identify non-aborting transactions and data dependencies leading to an abort decision.
Directory.open(id: Product) Directory.delete(id: Product)	Opens and deletes a transactional object, respectively. Used to extract data dependencies between objects.
HObj, HRef	Transactional objects and fields, respectively. Any references to these types flag the containing method as transactional (and therefore, of interest).
InTxn	Transaction context type. Same as above.

Table 1: Method invocations and reference types that aid in identifying transactional methods and their features (static analysis, first pass).

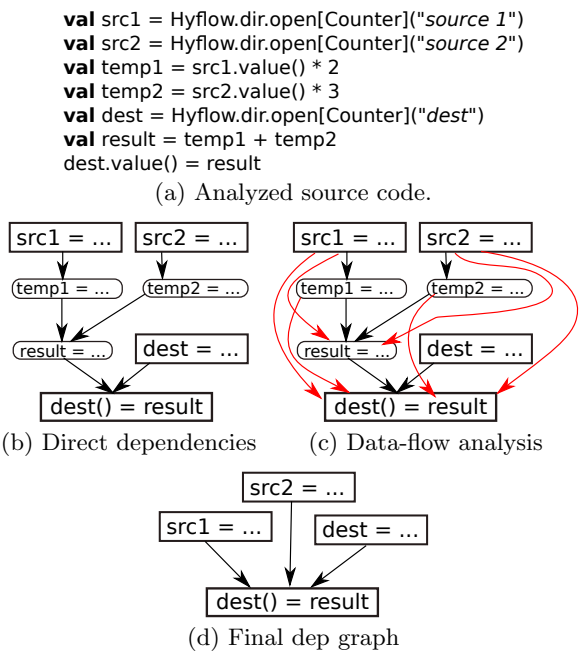


Figure 4: Forward data-flow analysis example for extracting the intra-method dependency graph. The rectangles represent nodes (units of interest), while the rounded rectangles are values.

4.4 Byte-code Rewriting and Trace Collection

Once all transactional method invocation sites are known and tagged, we rewrite the method byte-code to make certain information available at run-time. For every invocation of a transactional operation (object open, field read/write, etc.), we change the invocation to a different method that acts as a wrapper around the desired operation. This wrapper method takes an extra argument, the *tag id* (i.e., an invocation site identifier), which it logs before passing control to the transactional operation. The tag id is filled in by the byte-code rewriter, as an integer constant.

Other outputs from the static analysis process are the static dependency graphs for all the methods, the global static invocation graph, and a number of other details:

- Method Unique Identifier (MUID) for each transactional method.
- For each transactional operation invocation: tag id, type of operation.
- For each transactional method call: tag id, type of method call, MUID and name for the invoked method.
- For each type of transaction: transaction name, MUID for transaction entry point.

Next, a representative trace is collected by running the workload using the modified byte-code. This will result in a log of all the transactions executed, and within those, the important transactional operations. Log entries contain:

- Transaction id. Differentiates between multiple concurrent transactions.
- Operation name, such as *atomic* (transaction request), *txn begin/commit/abort*, *obj create/open*, *field read/write*.
- Tag id. Identifies the static invocation site that generated this log entry. Available for *txn abort*, *obj create/open*, *field read/write*.
- Operation specific data. Generally, this is the run-time object id this operation acts upon. For *atomic* and *txn begin*, this is a string representing the transaction type.

5. GRAPH REPRESENTATION AND PARTITIONING

Once a trace is available, it is parsed and converted to a graph where nodes represent objects and edges represent transactions, as described in Section 2.2. A number of heuristics limit the size of the graph, such as object and transaction sampling, and coalescing the nodes that are accessed by the same set of transactions. Edges are assigned weights such that the resulting partitioning is optimized.

5.1 Edge Weights

We now explain the process of assigning edge weights. We aim to satisfy several conditions and optimization criteria:

- Due to the Granola transaction model, we can not easily allow data dependencies between partitions. Make a best effort attempt not to allow such dependencies.
- When possible, favor independent transactions to coordinated transactions.
- Favor single-node transactions to any kind of distributed transactions.

To satisfy the first rule, we assign the highest weights

(a) Dependency graph (b) Resulting partitioning graph

Figure 5: Example partitioning graph for a transaction with data dependencies.

to all edges that connect objects having data dependency relationships with each other (*heavy edges*). For example, in Figure 5(a) we show the static dependency graph for a transaction. Nodes *1*, *2*, *3* and *4* represent static invocation sites for some transactional operations. At run-time, one execution of this transaction uses objects A, B, C and respectively, D, at the four static invocation sites. The system would assign the following *heavy edges*: A-B, A-C, A-D and C-D (Figure 5(b)). They denote the A $\rightarrow$ B dependency, and the A $\rightarrow$ C $\rightarrow$ D chain.

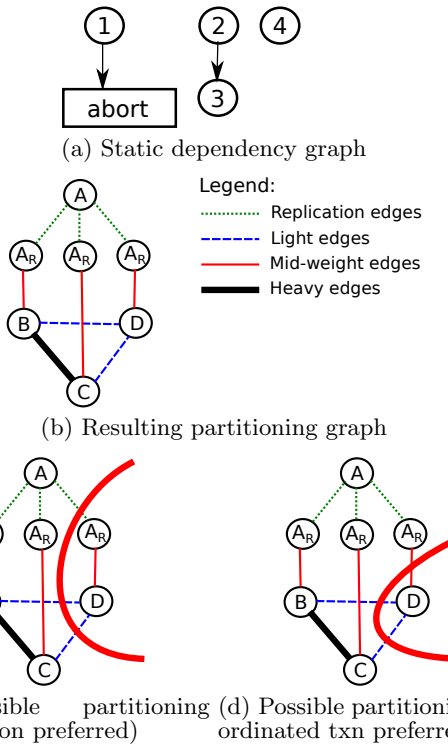
In our current implementation we use a very high weight (10,000) for heavy edges, effectively enforcing that no such edges will be broken. We should note that, with the Granola repository in locking mode, accessing remote objects would be possible, but with a penalty in performance. As such, instead of making heavy edges unbreakable, we could let the optimization process figure out if it may be, in fact, more desirable to break a small number of heavy edges instead of breaking a larger number of lighter edges. Thus our process could be extended with a heuristic that assigns weights to heavy edges based on the workload characteristics, instead of using a large constant as we do now.

The second rule refers to independent transactions as compared to coordinated transactions. These two models differ in that coordinated transactions execute a two-phase commit round, and thus allow reaching the commit/abort decision based on data not available at all repositories. Independent transactions can be used when the transaction does not need to abort, or reaches a commit/abort decision based on data available to all participating repositories.

To encode this in the partitioning graph, we first identify abort operations. If a transaction does not have any abort operations, it may be executed using the independent transaction model. Thus all remaining edges in such a transaction receive the lightest weight possible (100, we call these *light edges*). On the other hand, if a transaction does have abort operations, we want to encourage replication of all objects that were used in the commit/abort decision, as opposed to engaging in a coordinated transaction. Thus we use a medium weight (500) for the edges that connect to all such objects. We call these *mid-weight edges*.

This use-case may lead to replicating an object, even if the object is only accessed by one transaction. This behavior is new to our work, and requires an adjustment to Schism's handling of replicated nodes, which was described in Section 2.2. Previously, a replicated node for object A was created for each transaction that accessed object A. With our use-case, it is possible that more than one replicated node is required for the same transaction. This applies for the objects that lead to a commit/abort decision and may be replicated internally.

To better explain this behavior, we provide an example



(a) Static dependency graph

(b) Resulting partitioning graph

(c) Possible partitioning (replication preferred)

(d) Possible partitioning (coordinated txn preferred)

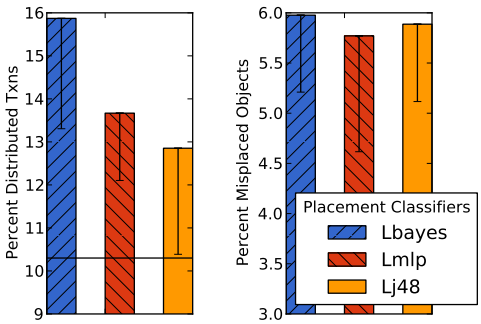
Figure 6: Partitioning graph example in the presence of aborts. The correspondence between the static invocation sites and objects accessed at runtime is: 1-A, 2-B, 3-C, 4-D.

in Figure 6. The static dependency graph is shown in Figure 6(a). The transaction makes an abort decision based on an object opened at invocation site *1*. Separately, it accesses three more objects (at sites *2*, *3* and *4*), with a data dependency between sites *2* and *3*. Assuming at run-time, the objects accessed are A, B, C and respectively, D, this transaction translates to the partitioning graph shown in Figure 6(b). Object A has three replica nodes, one for each other object in the transaction, arranged in a star-shaped configuration. The cost of replication edges are determined based on access patterns to object A throughout the workload. Because object A is used to make a commit/abort decision, its replicas connect to the other objects in the transaction using mid-weight edges $A_R - B$, $A_R - C$, and $A_R - D$. The other edges are heavy or light edges, based on the existence of dependency relationships.

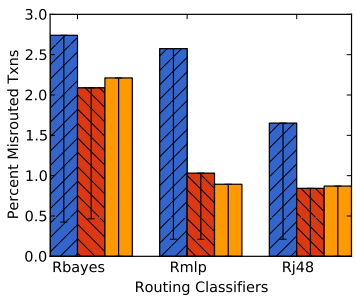
Two possible partitioning schemes are shown in Figures 6(c) and 6(d). In Figure 6(c), object D and one replica of object A are separated from the rest of the objects. This may happen, for example, when object A is rarely written to, and the cost of replicating it is therefore low. In this case, the transaction runs as an independent transaction. Alternatively, Figure 6(d) shows a partitioning scheme where only object D is separated from the others. There is no replication of object A, but the transaction must be coordinated.

5.2 Partitioning and Explanation

Once weights are assigned, we let METIS solve the optimization problem and propose a partitioning scheme. The result is a fine-grained association from objects to partitions.



(a) Distributed Txns (b) Misplaced Objects



(c) Misrouted Transactions

Figure 7: Results for a workload configured with 3 warehouses, with different classifiers. The trace used contains approx. 1200 transactions. Boxes show average values. Black markers show best classifier. Horizontal line in 7(a) shows theoretical best.

This can be used as-is only for small workloads. Specifically, we can not use object sampling to keep the problem size small, because the system would not know what to do with objects that do not appear in the mapping. If the problem size increases too much, running time and memory requirements rapidly increase as well.

We thus employ an explanation phase, where we train machine learning classifiers (using the Weka library) based on the fine-grained mapping. As opposed to Schism, we do not need to restrict our classifiers to be rule-based. Instead, we can use any classifier that works best for the current workload. This is possible because we have the whole stack under our control, and thus we do not need to restrict ourselves to what could be encoded efficiently in SQL. Although the current prototype hard-codes a single classifier type, we envision training a forest of classifiers in parallel, and choosing the ones that produce the best end-to-end results.

We train one classifier for each different type of objects. As in Schism, we use virtual partition numbers to represent replicated objects. For example, if there are two partitions in the system, P=1 and P=2, we use P=3 as a virtual partition to represent objects replicated on both partitions.

6. TRANSACTION ROUTING

Our system uses a stored procedure execution model, invoking transactions using the transaction's name and a list of arguments. Not knowing in advance the data each transaction is going to access makes it difficult to determine the

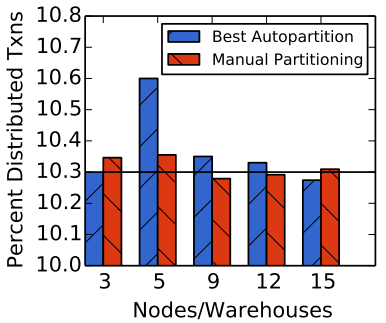


Figure 8: Best partition quality with increasing number of warehouses (lower is better). Horizontal line represents theoretical minimum. Differences are minimal (less than 0.3%)

partitions each transaction needs to be routed to. Using a simple directory based approach would be impossible. In Schism, the data a transaction will access is essentially known in advance — one looks at the `WHERE` clause of the SQL query for a quick decision about where to route transactions. This approach does not work in our situation.

Instead, we need to establish a link between a transaction's input arguments and the set of partitions it needs to be routed to for execution. For this, we again turn to machine learning, and employ another set of Weka classifiers. We train these routing classifiers using a workload trace. For each transaction in the trace, we want to route to at least the following partitions:

- Partitions that replicate any object in the write-set.
- A minimal set of partitions R, such that for any object X in the read-set, at least one partition $P \in R$ replicates object X.

Finding R is known as the *hitting-set problem*, which is NP-complete. Algorithms exist that approximate R, but are exponential in time [2]. We compute an approximation of the set R using a simple heuristic (Greedy), and we use that approximation to train our classifiers. This will be the output of the classifier. The input to the classifier is the list of arguments being passed to the transaction.

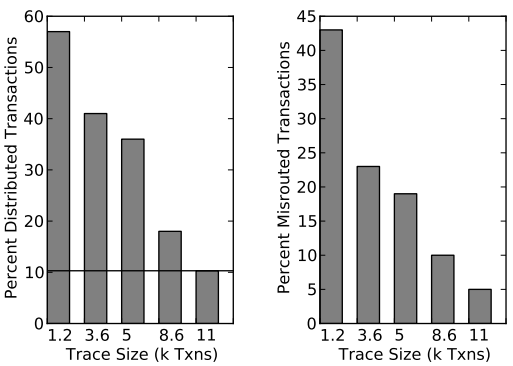
In our current implementation, we let the clients route transactions as they issue them. This is acceptable in a DTM environment where clients and servers are co-located. If clients can not be trusted with the identity of the servers, or the servers are located behind a firewall, it would be possible to employ a dedicated router/gateway process.

Classifiers do not always yield 100% accuracy. Misclassification at the routing stage may mean more nodes are contacted than strictly necessary, which is a benign situation. However, it is also possible that not enough nodes are contacted to allow completing the transaction. In such a situation, the transaction should abort on all currently participating nodes, and restart on a superset of the nodes. Algorithm 2 describes how to handle this situation (our prototype does not implement this mechanism yet).

The primitive to be used when executing a transaction is decided after the transaction has been routed. If only one repository is involved, the single-repository model will be chosen. For distributed transactions that do not explicitly abort (as identified in the static analysis phase) the inde-

Algorithm 2 Proposal for handling misrouted transactions.

N_{CRT} = the set of nodes participating in this transaction
upon open(X) = failed do
    ▷ find N_{REPL}, the set of nodes that replicate object X
    $N_{REPL} \leftarrow$ placement classifier (X)
    if $N_{REPL} \cap N_{CRT} \neq \emptyset$ **then**
        return ▷ X can be processed on a different node and
                ▷ the transaction can continue normally
    end if
    if current txn may write to X **then** ▷ from static analysis
        Restart txn on $N_{CRT} \cup N_{REPL}$
    else
        Restart txn on $N_{CRT} \cup ANY(N_{REPL})$
    end if



(a) Partitioning (b) Routing

Figure 9: Quality of partitioning and routing with respect to increasing trace size. (15 warehouses). In 9(a), horizontal line represents best known manual partitioning.

pendent transaction model is chosen. All other transactions use the coordinated model. This approach can be further refined by determining whether the decision to abort is made based on data available at all nodes. If so, an independent transaction can be used.

7. EVALUATION

We evaluate our partitioning process using TPC-C [7], a popular On-line Transaction Processing (OLTP) also used in other significant recent works [30, 15, 8]. While these works assume optimal manual partitioning, we employ our system in order to automatically derive a partitioning scheme. TPC-C emulates an order processing system for a wholesale supplier with multiple districts and warehouses. The workload was configured with between 3 and 15 warehouses. Throughput measurements were obtained on FutureGrid [11] with up to 15 virtual machines. Each node is an 8-core 2.9GHz Intel Xeon with 7GB RAM.

We used three classifier types (Naive Bayes [14], Multilayer Perceptron [13] and C4.5 decision trees [25]) for both object placement and transaction routing. Figure 7 shows results for a sample TPC-C workload. In this workload, approximately 10.3% of all issued transactions span more than one warehouses. These transactions would be executed as distributed transactions under the best known manual partitioning for TPC-C, i.e., each warehouse in its own partition,

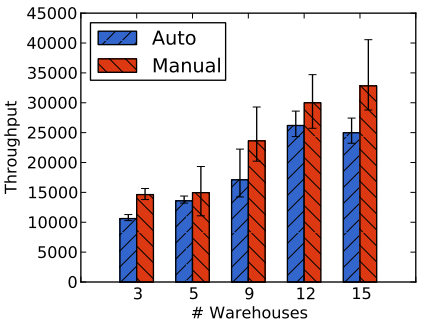


Figure 10: Total throughput with increasing number of nodes/warehouses and 10.3% distributed txns. The bar is the average value. The lower error bar is the standard deviation. The upper error bar is the maximum value.

and all *item* objects replicated at all partitions. We find that (for this workload) using C4.5 decision trees for placement and routing gives the best results, both in terms of minimizing distributed transactions and in terms of avoiding misrouted transactions.

Our systems proposes high quality partitions. By manual inspection of the resulting decision trees, we determined that many of our best partitions were identical to the best known manual partitioning scheme for TPC-C. The same conclusion is also supported by Figure 8, which compares the ratio of distributed transaction between our best partitions and the optimal manual partitioning, as the data size (number of warehouses) is increased.

We scope out a direct comparison against Schism — both our system and Schism essentially propose the same partitions (optimal) on TPC-C. Unlike Schism, our system is able to use the independent transaction model for all distributed transactions in this workload. Instead, Schism would use all 2PC-coordinated transactions, leading to lower performance. A direct comparison would only serve to showcase the differences between independent transactions and 2PC-coordinated transactions, and was done elsewhere [8].

Due to the random sampling of tuples and transactions, not every partitioning attempt had the same optimal result. This can be observed in Figure 7(a), where the best cases match the theoretical minimum of distributed transactions, but the average case is a few percentage points away. Several of the trained classifiers managed to reach 100% routing accuracy on our testing set, as seen in Figure 7(c). To deal with the inherent variability of random sampling, we recommend repeating the partitioning process several times (possibly in parallel) and choosing the best result.

As the data size is increased, however, the size of the trace that is the input to the system must also increase, otherwise the partition quality decreases. For example, if 3 warehouses only needed a trace with 1.2k transactions to give good partitions, 7 warehouses required 3.5k transactions and 15 warehouses needed 11k transactions. Figure 9 shows how the quality of partitioning and routing evolves with increasing the trace size, for 15 warehouses. In practice, one would likely start with a short trace (which can be evaluated faster) and progressively increase the trace size until the partition quality stops improving.

Tuple-level sampling rate	Creating graph from txn trace	METIS partitioning	Train placement classifiers	Compute partitions & train routing classifiers
5%	1m56	26s	22s	2m51s
10%	3m55	1m01s	37s	7m30s
20%	9m49	1m44s	1m02s	6m18

Table 2: Per phase running time, with 15 warehouses and a 89MB input trace containing 42k transactions.

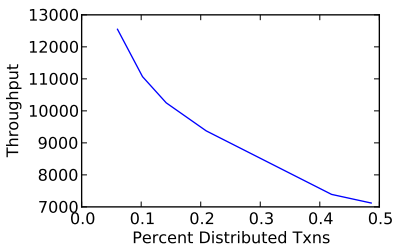


Figure 11: Total transactional throughput (3 warehouses), with a varying fraction of distributed txns.

To show how our process scales as we increase the graph size, we present running times for the various phases in Table 2. We varied the graph size by adjusting the tuple-level sampling factor (i.e., the ratio of data items present in the transaction trace that we represent as nodes in the graph, the remaining data items are ignored). We notice that a majority of the time is spent in the graph representation and evaluation phase. In the evaluation phase, most time is spent computing routing information for each transaction in the input trace (training the routing classifiers is relatively fast). We believe these two most time-consuming operations could benefit from further optimization.

Figure 10 shows transactional throughput measurements, compared to manual partitioning and routing. Experiments were allowed sufficient time for warming-up before measurements were started. Data points represent the average across eight measurements, and also relay standard deviation and the maximum value. We observed that enabling automatic routing and partitioning lead to 9-27% slow-down. The CPU time spent in additional code was measured to be negligible, and the total CPU load is light. By recording the execution time of the various routines and the communication latency, we observed the standard deviation becomes disproportionately larger than the average. This indicates the presence of large periodic breaks. It makes us believe the loss in performance is an indirect effect of the increased garbage collector (stop-the-world) activity caused by garbage generation in the classification code. This can be solved by careful memory optimization (e.g., object pooling), or using a different machine learning library.

We additionally varied the fraction of distributed transactions in a TPC-C workload to simulate the effect of partition quality has on throughput. Results are shown in Figure 11. Fewer distributed transactions clearly lead to better performance. It is noticeable how the effect is strongest when distributed transactions account for less than about 10-15% of the total workload. Thus, optimizing the quality of partitioning can bring large benefits and is especially important for workloads with less than 10-15% distributed transactions.

8. RELATED WORK

In the last decade, several proposals for scalable transactional storage [10, 1, 5, 3] are presented. Some of them target large scalability relaxing strong consistency [10, 5] ensuring respectively eventual and timeline consistency. Megastore in [3] is designed for very large scale on the Internet and it is based on state machine replication. Sinfonia [1] is similar to Granola but it requires a-priori knowledge of lock-set and it does not support independent transactions.

In context of DTM, a number of papers recently appeared [23, 24, 21, 27]. They provide new protocols optimizing particular scenarios but none of them reaches performance comparable to Granola. Additionally, some of them are based on partial replication where data is always stored manually over the nodes without exploiting any automation that allows optimizing the application access pattern. Our new automatic framework for partitioning data, although it is suited for the Granola [8] model, can be adopted (partially or totally) by any of previous published works for improving the locality of transactional accesses.

Partitioning techniques have been widely studied in context of DBMS where the typical approach is to enumerate possible partition schemes and evaluate them using different methodologies. In [31] the authors propose a stochastic approach for clustering data in object oriented DBMS. In context of distributed storage systems, in [4] and [5] are proposed systems acting with continuously re-partition data to increase the balancing. Unfortunately these strategies cannot be easily ported in transaction processing due to the presence of incoming transactional requests. AutoPart [22] is an automated scheme designed for multi-terabyte datasets, without any OLTP requirements. A dynamic vertical partitioning approach based on query patterns was recently proposed in [26]. However it is better suited for applications where such information does not tend to change over time. Autoplacer [20] approaches data placement in distributed key-value stores as an optimization problem.

9. CONCLUSION

We have developed a methodology for using automatic data partitioning in a Granola-based Distributed Transactional Memory. We perform static byte-code analysis to determine transaction classes that can be executed using the independent transaction model. We also use the analysis results to propose partitions that promote independent transactions. Due to our DTM focus, we take a machine-learning approach for routing transactions to the appropriate partitions.

10. ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation under grant CNS-1217385.

11. REFERENCES

[1] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM TOCS '09*.

[2] Amir and Farrokh. New approaches for efficient solution of hitting set problem. NASA JPL, 2004.

[3] J. Baker, C. Bond, J. Corbett, JJ Furman, A. Khorlin, J. Larson, J. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR'11*.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. OSDI '06.

[5] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *VLDB '08*.

[6] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *OSDI '12*.

[7] TPC Council. "tpc-c benchmark, revision 5.11". Feb 2010.

[8] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. USENIX ATC'12.

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *VLDB 10*.

[10] G. DeCandia, De. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *SOSP '07*.

[11] Geoffrey Fox, Gregor von Laszewski, Javier Diaz, Kate Keahey, Jose Fortes, Renato Figueiredo, Shava Smallen, Warren Smith, and Andrew Grimshaw. *FutureGrid - a reconfigurable testbed for Cloud, HPC, and Grid Computing*. CRC Computational Science. Chapman & Hall, 04/2013 2013.

[12] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. SIGMOD '08.

[13] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1998.

[14] George John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *In Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345. Morgan Kaufmann, 1995.

[15] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *VLDB'08*.

[16] G. Karypis and V. Kumar. Metis - serial graph partitioning and fill-reducing matrix ordering, version 5.1, 2013.

[17] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[18] P. Lam, E. Bodden, O. Lhotak, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS '11*.

[19] Hall M, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: An update. In *SIGKDD Explorations*, 2009.

[20] João Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodriguesauto. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. In *ICAC '13*.

[21] R. Palmieri, F. Quaglia, and P. Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *SRDS '11*.

[22] S. Papadomanolakis and A. Ailamaki. Autopart: automating schema design for large scientific databases using data partitioning. In *SSDBM '04*, 2004.

[23] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware*, 2012.

[24] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.

[25] Ross Quinlan. C4.5: Programs for machine learning. In *Morgan Kaufmann Publishers, San Mateo, CA.*, 1993.

[26] L. Rodriguez and XiaoOu Li. A dynamic vertical partitioning approach for distributed database system. In *SMC '11*.

[27] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS '10*.

[28] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

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

[30] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[31] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. SIGMOD '91.

[32] A. Turcu and B. Ravindran. Hyflow2: A high performance distributed transactional memory framework in scala. In *PPPJ 2013*.