Automated Data Partitioning for Independent Distributed Transactions

Alexandru Turcu
Virginia Tech
talex@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Granola is a recently proposed transactional execution protocol that employs a novel timestamp-based synchronization for executing certain classes of distributed transactions. However, Granola has two critical drawbacks. *A)* It requires users to manually define a data partitioning scheme and choose the appropriate transaction primitive. We seek to automate this process. We employ an existing graph-based algorithm (Schism) for partitioning transactional data, and extend it to be compatible with the additional insights and requirements of the Granola protocol. *B)* Granola requires a-priori knowledge of data location for routing transactions to repositories. We develop a routing mechanism based on machine learning to overcome this issue.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; H.2.4 [*Systems*]: Transaction processing

Keywords Data Partitioning, Transactional Data Store

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.

Cowling and Liskov in [1] recently proposed a new model for scalable distributed transactional storage, called Granola, in which transactions are processed locally and only one round of communication between the user and system is needed. Additionally, this approach exploits the presence of multiple machines, where possible, to process independent portions of the same transaction in parallel on different nodes, in order to overlap their computation (named *independent transactions*).

However, even though the Granola model is appealing, its assumptions restrict its applicability mainly because: *i)* data needs to be well partitioned such that transactions can be executed according to the *independent transaction* paradigm, namely split into multiple execution flows running in parallel on different partitions; and *ii)* users executing the transactions require a-priori knowledge of partitions accessed by submitted transactions for contacting the right nodes. This model, as is, cannot be considered as a transparent framework for the application developer because it requires significant manual effort - the developer must understand the application business logic and define partitions manually.

Motivated by these problems, we present an automatic framework to enable the exploitation of advantages offered by the Granola model on general-purpose distributed transactional applications. Specifically, we extend an existing graph-based data partitioning algorithm, Schism [2], to be compatible with the additional insights and requirements of the Granola model (Section 2). Additionally, we solve the problem of routing transactions, developing a new efficient mechanism based on machine learning that overcomes the lack of knowledge about the nodes storing objects. We also provide simple transactional abstractions to the programmer for interfacing with the distributed system, making our proposal easily pluggable for existing applications (Section 3).

Preliminary results of the framework show its capability to organize data for maximizing the throughput of transactions running in independent mode. Transactional throughput on TPC-C is comparable to the original Granola, where partitioning and routing are done manually.

2. Partitioning Process

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

Middleware 2013 Posters and Demos Track, December 9-13, Beijing, China.

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 unique identifier of the operation, as was tagged in the previous step.

The next three phases are similar to the corresponding phases in Schism [2]. The *graph representation phase* processes the previously collected workload trace and creates a graph where nodes represent objects and edges represent transactions. This graph is governed by the same rules as in Schism. Additionally, edge weights are updated to reflect the new transaction models, along with their restrictions and desirability. The graph is then partitioned using METIS [3] in the *partitioning phase*. The result from this step is a fine-grained association from object identifiers to partitions. Next, a concise model representing 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 depending on the number of partitions it executes on, it's need to abort, and whether it writes any objects (or is read-only).

3. 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. Evaluation

Our implementation is based around Hyflow2 [4], a JVM-based DTM framework written in Scala. We evaluate our partitioning process using TPC-C. The workload was configured with between 3 and 15 warehouses, and traces of different lengths were obtained.

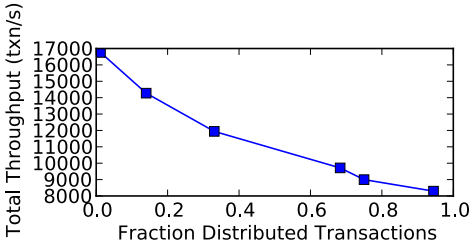

Figure 1. Transactional throughput, with 3 warehouses.

Figure 1 shows transactional throughput of a 3-warehouse workload, partitioned across three repositories. Throughput is high, comparable with the original Granola implementation, where partitioning and routing is done manually.

Acknowledgements

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

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

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

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

[4] A. Turcu, B. Ravindran, and R. Palmieri. Hyflow2: a high performance distributed transactional memory framework in scala. In *PPPJ '13*.