Gyro: A Modular Scale-Out Layer for Single-Server DBMSs

Habib Saissi
TU Darmstadt
saissi@cs.tu-darmstadt.de

Marco Serafini
University of Massachusetts Amherst
marco@cs.umass.edu

Neeraj Suri
TU Darmstadt
suri@cs.tu-darmstadt.de

***Abstract*—Scaling out database management systems (DBMSs) requires distributed coordination, which can easily become a bottleneck. Recent work on speeding up distributed transactions has addressed this problem by proposing scale-out techniques that are deeply integrated with the concurrency control mechanism of the DBMS.**

This paper explores the design of *modular* coordination layers, which encapsulate all scale-out logic and can be applied to scale out any unmodified single-server DBMS. It proposes Gyro, a modular coordination layer that runs on top of a collection of single-server DBMS instances and interacts with them only through their client interface. Gyro distributes the load by ensuring that as many requests as possible are executed by only one DBMS instance. Our experiments show that modular distributed coordination is practically viable and can be much faster than traditional distributed transaction protocols using two-phase commit.

I. Introduction

Online transaction processing (OLTP) applications, such as online shopping services, bidding services, or social networking systems, need scalability to support demanding workloads. These applications store their persistent state in a database management system (DBMS) that supports transactions with strong consistency guarantees. When the application load exceeds the capacity of a single DBMS server, it is necessary to scale out to multiple servers. This is typically achieved by partitioning the database and using distributed transactions.

Standard protocols for distributed transactions lock data at different servers until lengthy cross-server coordination is completed, creating a major bottleneck. This motivated a large volume of work on speeding up distributed transactions while preserving strong consistency [31], [8], [17], [27], [23], [11], [2]. This work proposes new DBMS designs where the concurrency control mechanisms for single-server transactions and distributed transactions are deeply intertwined. This makes it hard to port scale-out mechanism across different DBMSs and to add scale-out support to existing DBMSs.

In this paper, we explore a fundamentally different approach: is it practically viable to encapsulate all scale-out logic into a coordination layer that is separate and independent from the rest of the DBMS? Such a *modular* coordination layer should run on top of a collection of single-server DBMS instances. Each DBMS instance supports single-server transactions and executes transactions independently; it does not support distributed transactions or any other scale out mechanism to coordinate with other instances. The coordination layer can interact with DBMS instances only through their standard client interface. Therefore, the coordination layer can control *when* a transaction is submitted to *which* DBMS instance, but not *how* the transaction is executed. In addition, the coordination layer has no knowledge of the current state of the DBMS: when deciding where to send a transaction, it only has limited information, namely the type of the transaction and its input parameters. Finally, a truly modular solution should not require modifications to the application logic running on top of the DBMS, although it may analyze and instrument it.

A modular design like the one described above has many benefits over a monolithic DBMS design. Recent hardware innovations such as new multi-core and parallel architectures, in-memory computing, non-volatile memory, and novel storage systems, have lead to the design of new DBMSs, whose concurrency control mechanisms are optimized for single-server performance and do not readily support distributed transactions (see for example [25], [29], [26]). A modular coordination layer can scale out these DBMSs without modifying them or redesigning their efficient local concurrency control mechanisms. Modularity also enables picking the best scale-out mechanism for the application at hand without modifying the DBMS or the application itself.

In this paper, we show that a modular scale-out approach is practically viable. We introduce Gyro, a modular coordination layer that can scale out unmodified single-server DBMSs while guaranteeing serializability [19]. In order to effectively scale out, Gyro executes as many client requests as possible at only one DBMS instance. Determining which requests can be executed locally by which instance requires analyzing the application

Research supported, in parts, by EC H2020 Project CONCORDIA GA 830927

code. Gyro comes with a request classification tool that automates the classification of requests for standard OLTP applications that use simple WHERE clauses. The tool is sufficient to automate the analysis of the benchmarks we considered, TPC-W and RUBiS.

Our Gyro implementation runs Java applications running on top of unmodified JDBC-compatible databases. We used Gyro to scale out unmodified reference implementations of TPC-W and RUBiS. In both workloads, most transactions can be executed by a single DBMS instance. In a LAN setup, where all servers are running within one datacenter, Gyro increases maximum throughput by 4.2x and decreases minimal latency by 58.6x compared to MySQL Cluster, a popular system that layers two phase commit on top of an existing DBMS (MySQL). This is remarkable since Gyro provides a significantly stronger consistency guarantee (serializability instead of read committed isolation) and it interacts with MySQL instances solely through a JDBC client interface. In a WAN (i.e., geographically distributed) setup, scaling out from one to five locations using Gyro reduces latency by up to 47.9x and increases throughput by up to 2.8x.

Overall, we make the following contributions:

- We present Gyro, a distributed coordination layer that adheres to the modular scale-out paradigm;
- We propose sound criteria and an automatic classification tool to partition a workload using a modular coordination layer while preserving consistency;
- We show that modular coordination is a practically viable approach: using Gyro to scale out the MySQL DBMS in a significant improvement over its standard two-phase commit layer, MySQL Cluster.

II. Related Work

Distributed Transactions. The problem of designing modular concurrency control is complementary to much recent work that speeds up serializable distributed transactions. Callas [27] provides different degrees of isolation to different group of transactions based on their need. Tebaldi [23] combines different concurrency control mechanism in a single system. These systems optimize not only distributed but also single-server concurrency control. TAPIR [30] combines replication and distributed transactions in one protocol. Yesquel [2] introduces a novel distributed tree data structure. FaRM [11] uses RDMA to speed up transactions. Transaction chopping [22], [31], [17] modifies applications by chopping its transactions into sub-transactions. It assumes that the application can be chopped such that there are no "SC cycles" and only the first sub-transaction can abort and rollback. Gyro does not impose these restrictions.

Percolator [20], Omid [13], and Calvin [24], implement concurrency control and transactions on top of basic key-value stores, as opposed to single-server DBMSs with transactional support, and unlike Gyro implement distributed transactions. ElasTraS [10], G-Store [9], and MegaStore [4] only support ACID transactions within a single partition and do not offer full transactional support.

Weakly consistent scale-out. Recent work proposes strengthening weak consistency with *invariants*, like in the Red/Blue model [15], the Explicit Consistency model [5], and Invariance Confluence [3]. Requiring developers to define good invariants is challenging. Also, even with invariants, the system will still show a weakly-consistent behavior that would not occur in a sequential execution. By contrast, Gyro supports strong consistency.

Treaties. Treaties have replicas split the value of a certain numerical field and share the splits. Treaties apply to applications that make small commutative modifications to a shared global quantity at different replicas. Examples of treaties are the escrow protocol [18], the demarcation protocol [6], Homeostasis [21], and time-limited warranties [16]. Work related to the idea of treaties has also investigated relaxed notions of consistency such as bounded inconsistency [28] or consistency rationing [14]. Gyro is more generic since it does not make assumptions on the application, as treaties do.

III. Gyro Overview

Gyro targets multi-threaded applications that store their state on a database management system (DBMS) and scales them out by running them on multiple instances. Clients issue requests to an instance of the application, which can run, for example, in an application server like Apache Tomcat. To serve a request the application invokes a sequence of queries, enclosed in a single transaction, on a DBMS instance. Gyro assumes that the DBMS guarantees serializability of concurrent transactions.

Gyro Architecture. Figure 1 shows an example deployment with two DBMS instances. The DBMS instances are stand-alone: they do not need to support distributed transactions or even to communicate with each other. Gyro guarantees that DBMS instances are consistent and ensures the serializability for all transactions executed in the system. Each application instance is automatically instrumented with an instance of Gyro at compile time. We refer to a **Gyro instance** as an application instance with its local Gyro instrumentation.

Before an application thread starts executing a client request (Step 1 of Figure 1), the instrumentation invokes Gyro, which checks whether the request is local or global (Step 2). If the request is *local*, the application thread

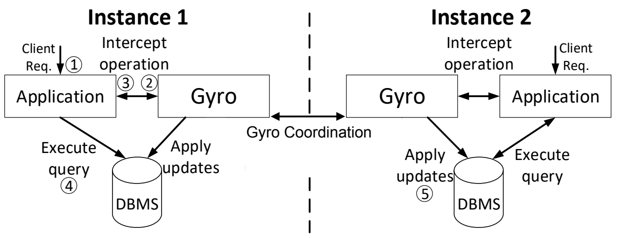


Fig. 1: Gyro system with two instances. The numbered arrows indicate the execution flow of *global* operations.

goes ahead and completes the request without further involvement by Gyro. Otherwise, Gyro holds up the *global* request by pausing the thread until it is allowed to proceed by Gyro's coordination protocol. At that point, Gyro gives back control to the application thread by resuming it (Step 3). Next, the application thread executes the global request on the DBMS (Step 4). Global requests are always executed by only one DBMS instance. Gyro instruments the application logic to record all updates executed on the DBMS by the global request. After the request has completed, the coordination protocol propagates these state updates to other instances. State updates received by a Gyro instance are applied directly on the local DBMS instance (Step 5). The state accessed by global requests is fully replicated.

IV. Request Classification

The key to scalability in Gyro is to maximize the number of local requests. We now discuss how Gyro automatically classifies local and global requests. Concretely, consider the example of an online shopping application with the following possible transactions:

```
createCart(cart_id);
addItem(cart_id, item_id, order_qty);
order(cart_id);
```

The transactions allow clients to create a cart, add items to the cart, and eventually proceed to checkout. Each cart and item is assigned a unique id. The `ITEMS` table associate each item id with its current availability in stock. Each row in the `SHOPPING_CARTS` table indicate a certain quantity of a item that has been added to a cart. The `createCart` transaction inserts a new cart row with id `cart_id` in the `SHOPPING_CARTS` table. The `addItem` transaction adds a row in the `SHOPPING_CARTS`. Before adding items to a cart, `addItem` checks in the `ITEMS` table whether the items are in stock. The `order` transaction proceeds to the checkout of a cart, again after verifying in the `ITEMS` table that all items in the cart are available in stock.

In the rest of the paper, we refer to the procedures the application offers to clients as **transactions**. Each transaction can have a certain number of input parameters. A **request** corresponds to a client request to execute the

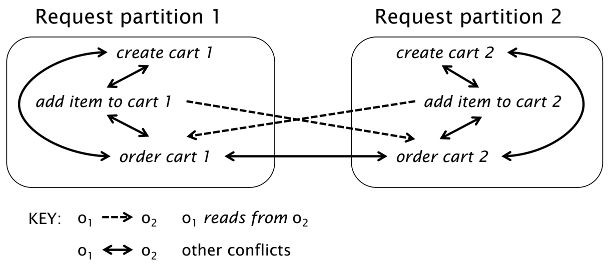


Fig. 2: Partitioning of the online store example. The *order* request is global, the other requests are local.

transaction with a set of concrete values for its input parameters. For example, a request can be the invocation of `order(5)` to order items in the cart with id 5.

In order to distribute load, Gyro maps each possible client request with a unique Gyro instance. It routes each request to the right Gyro instance based on the transaction and the input parameters invoked in the request (Step 1 in Figure 1). These predicates are determined through a process called **request partitioning**, which maps each request to a partition, which in turn maps to a single Gyro instance. The partitioning also indicates whether a request is global or local. Partitioning can be automated for typical OLTP applications, as discussed in Section VI, but it can also be manual. We will discuss how to characterize a correct request partitioning shortly.

In our example, a request partitioning could partition requests based on their `cart_id` parameter. We have one request partition per cart id. This ensures that all requests related to the same cart are routed to the same Gyro instance. Multiple request partitions (each for a different cart id) can map to the same Gyro instance. When a client request reaches the application, Gyro intercepts the request to determine whether it is local (Step 2 in Figure 1) based again on the request partitioning.

Correctness of Request Partitioning. Obtaining a request partitioning is an offline process. Determining the correctness of the partitioning requires the extraction of read/write sets. Conflict analysis looks at the write and read sets for each request in terms of rows of the database and groups them by transaction. For each transaction, conflict analysis considers an *over-approximation* of the write and read sets for *all* possible database states and *all* possible values of those input parameter that are *free*, i.e., not *bound* by the request partitioning predicate. In our online shopping example, the `cart_id` parameter is bound and used for the partitioning. Therefore, the `addItem` and `order` transactions need to consider all possible items that may be added to a particular cart, since `item_id` is a free input parameter.

The next steps is to analyze the conflicts among transactions. The result of conflict analysis for our online shopping example is depicted in Figure 2, which considers only two request partitions corresponding to two different cart ids. Requests in the same partition have write-write and read-write conflicts because they may all edit the same rows in the `SHOPPING_CARTS` table. Also, `order` transactions from the two partitions have read-write and write-write conflicts because they may all access the same items in the `ITEMS` table. An `order` transaction for cart 2 writes items into the `ITEMS` table that may be read by `addItem` for cart 1. So the execution of the `order` transaction modifies the behavior of the `addItem` transactions in other partitions. We say that the `addItem` transaction for cart 1 *reads from* the `order` transactions for cart 2. Note that `order` transactions do not read from `addItem` transactions in other partitions: an `addItem` transaction for cart 1 only writes data related to cart 1, which is not read by any `order` transaction for cart 2.

After performing conflict analysis, we can classify requests as local or global as follows:

***Definition** 1: A client request r is **local** iff no other request from a different request partition reads from r. A request that is not local is **global**.*

This correctness criterion can be applied to any request classification. In the example of Figure 2, `createCart` and `addItem` requests are local, while `order` requests are global. Note that requests for the same transaction can be classified differently based on their input parameters.

V. The Gyro Coordination Layer

Overview. Local requests can be executed locally by the DBMS instance they are mapped to since their updates are only visible to requests in the same partition. On the other hand, global requests require coordination among Gyro instances. First, they are executed by the instance they are mapped to (Step 4 in Figure 1) before their results are combined into a special write-only *state update* request that can be propagated to the rest of the instances (Step 5 in Figure 1). Propagating state updates instead of requests eliminates the need for propagating transactions that are local to other instances, even if they determine the behavior of global requests. Note that our description of Gyro's protocol omits the details of the state machine replication protocol used to make each Gyro instance fault tolerant where each Gyro instance replica is associated with a replica of the DBMS instance.

Coordination algorithm. Each Gyro instance (simply instance in the following) runs an instance of the algorithm (see Algorithm 1), which is executed in parallel by multiple threads. When an instance p receives a client request, it uses the classification criterion to determine whether the request is local (Step 2 of Figure 1). Local requests are executed by the local DBMS instance, using a standard client interface, and a reply is immediately sent back to the client without coordination (Lines 2-4 of Algorithm 1, Steps 3-4 of Figure 1). Global requests are *held up* at the application side until Gyro instances agree on a total order of execution.

There are several ways to agree on this total order. The current implementation of Gyro uses a protocol called Conveyor Belt, which is token-based and inspired by chain replication. Conveyor Belt passes a token among instances in a predefined order to ensure that global requests are totally ordered. At any time, only the instance holding the token is allowed to execute global requests. This instance is called primary. The other instances append the requests to a queue Q for execution at a later time (Line 6). Note that the queue Q must be thread-safe since Algorithm 1 is multi-threaded.

Upon receiving the token T, instance p invokes one $\textsc{receiveToken}(T)$ event at p, becoming the primary. Like any other event, a specific $\textsc{receiveToken}(T)$ event is handled in isolation by a single thread, while multiple other threads might be concurrently handling client requests. The token contains a sequence of tuples $\langle u, q \rangle$ where u is the state update request corresponding to a global request that has previously been executed by some instance q. As soon as an instance becomes primary, it executes all the state updates requests from other instances at the local DBMS (using its client interface) and removes its own updates as they have been already seen by all other instances (Lines 8-12, Step 5 of Figure 1). Note that if multiple instances map to the same DBMS instance, it is necessary to ensure that state update requests are applied only once. This is easy to guarantee since the order in which instances pass the token is fixed and globally known. Next, the primary executes the global requests that have been enqueued locally into Q. In order to ensure liveness, the primary copies an atomic snapshot Q' of the Q queue containing global requests submitted to p that have been waiting for execution (Line 13). This is because Q is concurrently modified by multiple threads that can submit global and local requests. Without copying an atomic snapshot, p might stay stuck executing incoming global requests in Q that are constantly being appended by other threads, and never give up the token. Then, p iterates over all global requests that have been pending up to that point in the Q' queue (Lines 14-18, Step 3-4 of Figure 1). The instance submits each request to the local DBMS instance, using its client interface. It then sends a reply r to the client

Algorithm 1: Conveyor Belt algorithm for instance p

```
upon receive $\langle$REQ, $o, c\rangle$ msg from client $c$ where
    if $o \in L_p$ then
        $r, * \leftarrow$ DBMS.execute($o$);
        send $\langle$REPLY, $r\rangle$ msg to $c$;
    else
        append $\langle o, c\rangle$ to $Q$;
upon event RECEIVETOKEN($T$)
    foreach $\langle u, q\rangle \in T$ do
        if $p = q$ then
            remove $\langle u, q\rangle$ from $T$;
        else
            DBMS.execute($u$);
    $Q' \leftarrow$ atomic-snapshot($Q$);
    foreach $\langle o, c\rangle \in Q'$ do
        $r, u \leftarrow$ DBMS.execute($o$);
        append $\langle u, p\rangle$ to $T$;
        send $\langle$REPLY, $r\rangle$ msg to $c$;
        remove $\langle o, c\rangle$ from $Q$;
    PASSTOKEN($T$);
```

c and appends the resulting state update request u to the token before removing the requests that have been appended. Finally, the instance gives up the primary role by calling PASSTOKEN(T) to pass the token to the next instance (Line 19).

For lack of space, we have included details about the implementation of the Conveyor Belt protocol and a formal correctness proof in the appendix.

VI. Automatic Request Classification Tool

Gyro includes an automatic request classification tool that analyzes the application code and outputs the classification criteria used by the Gyro runtime to classify requests as local or global and route them to the right instance. The tool applies to common OLTP applications using simple SQL queries with `WHERE` clauses in equality form and is sufficient to automatically process our two benchmarks, TPC-W and RUBiS, without modifications. For application with more complicated SQL queries, the request classification must be obtained manually.

In this section we show how we extract read and write sets from the source code and describe the automated partitioning algorithm which takes the read and write sets as inputs and generate an optimal request partitioning.

Identifying read and write sets. An OLTP application usually has a relatively small number of transactions, which can correspond to a huge number of possible requests. Therefore, the static analysis algorithm determines the read and write sets at the granularity of transactions. An entry e in either sets is a pair $e = \langle A, C\rangle$, where A is a set of *accessed attributes* and C is a *condition*.

The accessed attributes set in the read set contains all table attributes (i.e., columns) that are read and returned as output of the transaction. A write set contains all attributes that are updated by the transaction. The condition of a read or write set is the predicate used to select the specific rows in the table for which the attributes are modified.

The tool analyzes applications consisting of a set of transactions that access a database through SQL queries. For example, the `doCart` transaction in TPC-W updates a shopping cart with id `sid` by adding, removing or updating item with id `iid` in a quantity `q`.

```
doCart(sid, iid, q){
            ...(abridged code)...
            exec("UPDATE SHOPPING_CARTS
            SET QTY = q WHERE ID = sid
            AND I_ID = iid");
            ...(abridged code)...   }
```

The static analysis algorithm looks at *all* SQL statements contained in the transaction, regardless of the execution path. We used Java parser [1] to extract SQL queries and to map input parameters to the used query parameters. Each SQL statement corresponds to an entry in a read or write set. Consider for example the SQL statement highlighted in the pseudocode and rename the table `SHOPPING_CARTS` as `SC` for brevity. This statement corresponds to a write set entry e. The accessed attribute for e is specified in the `UPDATE` clause, so $e.A$ =`SC.QTY`. `INSERT` queues also correspond to entries in the write set and their accessed attribute is specified in the `INSERT` statement, while for read set entries the accessed attribute corresponds to the `SELECT` query. The condition of the entry corresponds to the content of the `WHERE` clause of the query, so in this case $e.C$ = (`SC.ID = sid` $\wedge$ `SC.I_ID = iid`). The condition binds the value of the input parameters of the transaction, i.e., `sid` and `iid` in this case, with the values of the table attributes of the specific rows for which the attributes in $e.A$ are accessed by the transaction, `SC.ID` and `SC.I_ID = iid` in our example.

Conflict detection phase. The partitioning algorithm is illustrated in Algorithm 2. The first phase of the algorithm is *conflict detection*, which looks at all pairs of transactions that have a conflict on some table attribute. A conflict between transactions occurs if some of the requests relative to these transactions can conflict. For each pair of transactions (t, t'), the algorithm builds a condition predicate $C_{t,t'}$, in disjunctive normal form, that expresses the condition that the values of the input parameters of t and t' must take so that a conflict occurs on the same row(s) of the same table(s). In other words, the condition characterizes the set of requests of the two transactions that are conflicting. If a conflict between the two transactions is possible, $C_{t,t'}$ is added to a set called *Conflicts*. Note that we also consider conflicts between two requests of the same transactions where $t = t'$.

Algorithm 2: Partitioning Algorithm.

```
input  : Set $T$ of transactions
input  : Read set $R_t$ and write set $W_t$ for each transaction $t$
output : Array $P$ of partitioning parameters $P[t]$ for each
         transaction $t$
// Conflict detection
foreach pair $t, t' \in T$ do
    $C_{t,t'} \leftarrow$ false;
    if $\exists r \in R_t, w \in W_{t'} : r.A \cap w.A \neq \emptyset$ then
        $C_{t,t'} \leftarrow C_{t,t'} \vee (r.C \wedge w.C)$;
    if $\exists w \in W_t, r \in R_{t'} : w.A \cap r.A \neq \emptyset$ then
        $C_{t,t'} \leftarrow C_{t,t'} \vee (w.C \wedge r.C)$;
    if $\exists w \in W_t, w' \in W_{t'} : w.A \cap w'.A \neq \emptyset$ then
        $C_{t,t'} \leftarrow C_{t,t'} \vee (w.C \wedge w'.C)$;
    if $C_{t,t'}$ is satisfiable then
        Conflicts $\leftarrow$ Conflicts $\cup C_{t,t'}$;
// Partitioning optimization
return $\min_P$ cost($P$, Conflicts);
// Estimate the volume of conflicts
function cost($P$, Conflicts)
    foreach $C_{t,t'} \in$ Conflicts do
        $k \leftarrow P[t]$;
        $k' \leftarrow P[t']$;
        foreach table attribute $A$ do
            remove all clauses $(k = A \wedge k' = A \wedge \ldots)$
              from $C_{t,t'}$;
        if $C_{t,t'}$ not satisfiable then
            remove $C_{t,t'}$ from Conflicts;
    return $\sum_{C_{t,t'} \in Conflicts} weight(t) + weight(t')$;
```

Let us consider again the TPC-W example. The `createCart` transaction creates a new row in the `SHOPPING_CARTS` table (`SC`) such that `SC.ID = sid`, where `sid` is the id of the shopping cart:

```
    createCart(sid){
        ...(abridged code)...
        exec("INSERT INTO SHOPPING_CARTS
        (ID) VALUES (sid)");
        ...(abridged code)... }
```

The write set of `createCart` contains entry $e = \langle$ `SC.ID` ,`SC.ID = sid` $\rangle$. Given the write set of `doCart`, we derive that there is a write-write conflict between the two transactions with condition $C_{t,t'}$:

$(\texttt{SC.ID = sid}) \wedge (\texttt{SC.ID = sid'}) \wedge (\texttt{SC.I\_ID = iid'})$

where `sid` is a parameter of `createCart` and `sid'` and `iid'` are parameters of `doCart`.

Partitioning optimization phase. The next phase is called *partitioning optimization* and it finds the request partitioning array P that minimizes global requests. The partitioning can reduce the cost of conflicts by mapping two conflicting requests to the same partition, and thus DBMS instance, such that the conflict becomes local.

The cost function finds out the potential a request partitioning has to eliminate conflicts. Consider two transactions t and t' that conflict, and let k and k' be the parameters used for their partitioning. Gyro uses the same deterministic routing function for all requests, so two requests with the same value of their partitioning parameters k and k' will be sent to the same Gyro instance. Therefore, all conflicts that arise because of a necessary condition $k = k'$ will be local to one DBMS instance, and they will not require global coordination. The most common case when this condition arises is when k and k' are used to identify a row based on the value of the same attribute A, so there is a clause in the conflict condition of the form: $(k = A \wedge k' = A \wedge \ldots)$.

In our TPC-W example, let P be a request partitioning array such that `sid` is the partitioning parameter for both `doCart` and `createCart` transactions. The conflict condition in the previous equation is of the form $(k = A \wedge k' = A \wedge \ldots)$, where k =`sid`, k' =`sid'`, and A =`SC.ID`. This condition is equivalent to saying that the conflict among the two transactions arises only if `sid =sid'`. The same deterministic routing function is used for both transactions, so conflicting requests will always be sent to the same DBMS instance. We can thus remove this conflict from the *Conflicts* set.

After removing all conflicts that become local thanks to the request partitioning array P, we estimate the cost of the remaining global conflicts by summing up the weight of the conflicting transactions in *Conflicts*. Estimates of the relative frequency of the transactions can be used for better cost estimations while assigning a weight of 1 to each transaction yields a minimal number of conflicts.

The algorithm finally searches for the request partitioning array that minimizes the cost. In the workloads we considered, as in most transactional workloads, the number of transactions and their parameters is not very large, so an exhaustive search of all possible partitionings to find the best one is feasible. However, the algorithm can also use more sophisticated search strategies.

VII. Evaluation

To evaluate Gyro, we designed three sets of experiments. First, we evaluate the fraction of local requests in a workload that is sufficient to see performance improvements with Gyro. Then, we compare Gyro against a traditional two-phase commit protocol. Finally, we analyze the performance of Gyro in a WAN setting.

A. Experimental Design

Experimental setup and the baseline. We implemented Gyro in Java (about 2k LOC). Each Gyro instance instruments a local application instance running on Tomcat, and interacts with the DBMS solely through a standard JDBC interface (see Figure 1). DBMS instances are stand-alone and do not interact with each other.

We conducted our experiments on resource-constrained Amazon EC2 instances. Beyond reducing the cost of running the experiments, this choice made it easier to

Application	Transaction classification			Read-only	Total
	L	G	L/G		
TPC-W	15	5	–	13	20
RUBiS	14	4	8	17	26

TABLE I: Request classification for the case studies: Number of local (L), global (G), and local/global (L/G) transactions. Some of these transactions are read-only.

saturate the system and thus show the relative speedup of scaling up. Therefore, our evaluation focuses on relative speedups, which measure the benefit of using Gyro, rather than on absolute performance figures, which would just measure the performance of the DBMS and the hardware configuration Gyro runs on top of. We run our experiments on Amazon EC2 using T2 Medium instances. These are among the most resource-constrained instances made available by Amazon. Each instance has 4 GB of RAM, two virtual cores and is equipped with an Amazon EBS standard SSD with a maximal bandwidth 10000 of IOPS. The instances run Ubuntu Server 14.04 LTS 64, MySQL 5.5.49-0 and Apache Tomcat 7.0.52.

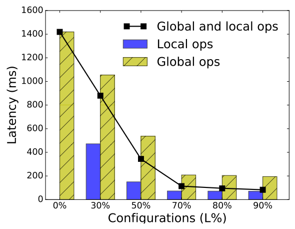
In the LAN experiments, all servers are located in the same site (datacenter) in Germany. For the WAN (geographically distributed) experiments, we place servers in five different sites to simulate a geographically distributed system. The sites are in Germany (G), Japan (J), US east (US), Brazil (B), and Australia (A). We add these locations in the aforementioned order. For example, a configuration with three locations consists of servers in G, J, and US. Table II reports the inter-site latencies among servers. The intra-site latency, relevant for the local setup, is about 20 ms. We used separate client instances with identical configurations as the servers and are located in the same sites. In the WAN setting, we use five client machines in every configuration, one for each location, and direct requests to the closest Gyro instance. We equally distribute client threads across the clients and measure end-to-end throughput and latency at client side in all our experiments.

Choosing benchmarks. We use a set of three benchmarks. The first is a synthetic micro-benchmark, where clients can be tuned to produce a specific mix of local vs. global requests. We use this benchmark to evaluate pure coordination cost, so transactions are void: they are processed by Gyro but not submitted to the DBMS. Then, we used two other benchmarks that are commonly used in related work such as [15], [31]: TPC-W, an on-line book store [7], and RUBiS, an auction website [12].

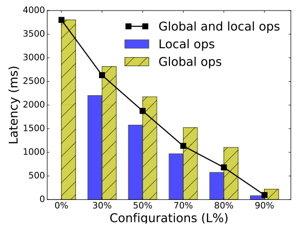
We applied our request classification tool on the code of these benchmarks without modifications. The results of the analysis is reported in Table I. For TPC-W, the local transactions mainly involve updating customer data,

Locations	G	J	US	B	A
Germany (G)	X	253ms	92ms	193ms	314ms
Japan (J)		X	153ms	282ms	188ms
United States (US)			X	145ms	229ms
Brazil (B)				X	322ms

TABLE II: Inter-site latencies among WAN locations.



(a) Non-saturated throughput.



(b) Saturated throughput.

Fig. 3: Micro-benchmark: latency of global and local requests with different frequencies of local requests.

and are partitioned by customer id, or manipulations of the shopping carts, and are partitioned by cart id. Global transactions involve ordering books or administrative transactions such as updating the books list. In Rubis, we use a double-key scheme, whereby many transactions are partitioned by both customer id and item id. If both parameters route to the same Gyro instance, the request is considered local, otherwise it is considered global. The local transactions involve the user browsing through his personal profile. Global transactions include a global search for items or browsing through a user's own bought items. Local/global transactions involve bidding, buying and selling, and may be local or global depending on the values of the input. Read-only transactions are local transactions that do not modify the database.

We use the default bidding mix with 15% actual bidding operations for RUBiS and the shopping mix with 30% ordering operations for TPC-W. Both workloads simulate a typical usage of the systems but still exhibit a considerable number of local operations that can be leveraged by Gyro.

B. Local vs. Global Request Ratio

The performance of Gyro depends critically on the fraction of local operation requests. To analyze the effect of different local operations ratios on Gyro we used a synthetic workload where we can precisely specify these ratios. The execution time of operations (global or local) is fixed to 5ms. We use a WAN setup to magnify the negative impact of global operations. We use clients at five locations, servers at three locations, and vary the percentage of local operations in the workload from 0% to 90%. As discussed in Section VII-A, our analysis focuses on relative speedups rather than absolute performance.

If the throughput of the system is not saturated, Gyro minimizes latency as long as the fraction of local requests

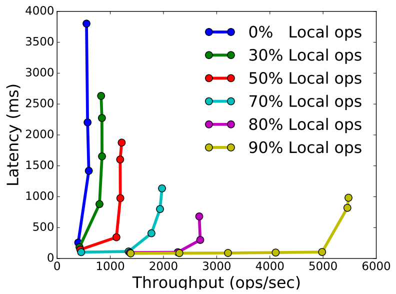


Fig. 4: Micro-benchmark: throughput and latency with different frequencies of local requests.

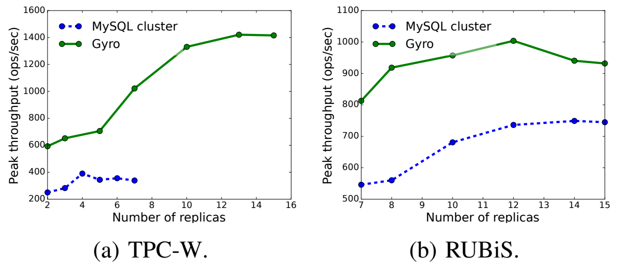
is 70% or higher (Figure 3a). In all our case studies, the fraction of global requests was below 30%. Local requests have very low latency in a non-saturated system since they do not require coordination. Global requests have much higher latency, more than 3x higher than local requests, since they must circulate between Gyro instances in three locations. But as long as we have many more local requests than global ones, the average latency is low and dominated by the latency of local requests.

Global requests saturate the system if they exceed 30% of the workload. Interestingly, it is not only the latency of global requests that grows; the latency of local requests also grows, albeit less significantly. This is because more and more global requests need to be executed at all instances, defeating the advantages of scaling out. The latency of global requests also starts dominating the mix as their relative frequency grows. This sensitivity to the rate of global operations is magnified when local requests alone are sufficient to saturate the system (Figure 3b).

In terms of maximum throughput we observe that, unsurprisingly, the performance of Gyro is highly sensitive to the fraction of local operations in the workload, even in the top range of the spectrum (see Figure 4). For instance, with a workload of 30% of local operations, the system starts to saturate already around 600 ops/sec, while in a workload of 90% local operations the saturation starts only around the 5477 ops/sec.

C. Gyro vs. 2PC

We now show the practical viability of our modular scale-out approach by comparing with a standard two-phase commit protocol. This protocol represent a common baseline for comparison across many scale-up algorithms. Since we use MySQL as reference DBMS, we consider MySQL Cluster as a target baseline. MySQL Cluster can only provide the *read committed* isolation level, whereas Gyro provides *serializability*, which is significantly stronger and more expensive to achieve. Nonetheless, Gyro is still able to achieve a large speedup over MySQL Cluster.



(a) TPC-W. (b) RUBiS.

Fig. 5: Scalability of Gyro vs. MySQL Cluster.

We choose a LAN setting for this experiments since it is more favorable for MySQL Cluster. Overall distributed transactions perform much better over LANs than over WANs. We also use the two standard benchmarks , TPC-W and RUBiS, described in Section VII-A.

MySQL Cluster requires users to specify how to partition data. For fairness, we have adopted the same data partitioning for both Gyro and MySQL Cluster. We first ran Gyro's request classification tool on the code of both benchmarks. From the request classification logic, we extracted the resulting data partitioning scheme and applied it to MySQL Cluster. Each server acts as a MySQL Cluster server and a data node that stores exactly one data partition. We additionally designate one server as the manager for the initial setup.

We examine the ability of both approaches to scale out. In this local setting, we intensify the workload by increasing the number of clients. In Figure 5 we show how the peak throughput develops while varying the number of DBMS instances (and thus of Gyro instances) in the system for TPC-W and RUBiS, focusing around the maximum of the curve. Peak throughput is defined as the maximum throughput a system can sustain while ensuring an average latency of less than 2000ms.

Figures 5a and 5b show the same trend for both TPC-W and Rubis: as the number of DBMS instances grows, the increased cost of distributed coordination eventually outweighs the gain of additional resources to run transactions that require no coordination. This upper bound in scalability represents the inherent cost of achieving strong consistency in the workloads we consider, which are not perfectly partitionable.

Both figures 5a and 5b show that Gyro scales much better than MySQL Cluster. This is remarkable given that Gyro only uses a standard client interface, and provides a stronger consistency guarantee (see Section VII-A). In the case of TPC-W we can see that while the performance of MySQL Cluster starts to degrade with configurations of more than 4 nodes, Gyro continues to deliver at a much higher throughput until it reaches a configuration of 13 DBMS instances. On the other hand, with the RUBiS workload, Gyro reaches a point of saturation at the same configuration, namely 12 DBMS instances, as MySQL Cluster but still consistently achieves higher throughput. Overall, Gyro outperforms MySQL Cluster both in terms of maximal throughput and latency by up to 58.6x for latency and about 4.2x for throughput in the case of TPC-W. For RUBiS, Gyro achieves a 1.4 maximal throughput speedup and reduces the latency up to 35.7x.

Gyro performs significantly better than MySQL Cluster because distributed transactions, which are used by the latter, lock rows *across multiple servers*. The necessary coordination with remote machines in MySQL Cluster prevents the progress of concurrent conflicting transactions that access the same rows while multi-server coordination is completed. In contrast, when a Gyro instance receives global operations that require remote coordination, it merely enqueues the operations until the token is received. This allows other concurrent local operations to make progress. Gyro holds local operations only for a short period of time when a server has the token and executes global operations. Otherwise, it solely resorts to the local DBMS locking mechanism for transaction concurrency.

TPC-W and RUBiS show different results due to different read-only operation ratios. In TPC-W many of the local operations are write operations that, in MySQL Cluster, involve distributed transactions. Therefore, TPC-W benefits tremendously from operation partitioning. The RUBiS workload contains more local operations, but a much larger fraction is read-only. RUBiS thus profits from the read-only transaction optimizations implemented by MySQL Cluster. These results highlight that existing DBMSs already require minimal coordination for read-dominated workloads, making Gyro a better fit for write-heavy, and thus hard to scale out, workloads.

D. Gyro on WANs

The previous experiments showed that Gyro outperforms a two-phase commit protocol even in a LAN setting, which favors 2PC. We now evaluate the performance of Gyro in a WAN (i.e., geographically distributed) setting, where coordination is even more expensive and scalability is more challenging. We use two baselines: (1) a single server deployment of standard MySQL (without Gyro) **(centralized)**, and (2) Gyro optimized to run only read-only requests as local, while all the rest are global **(read-only)**. Read-only optimizations are common in many systems. We show that the performance benefits of Gyro go well beyond a simple read-only optimization.

Latency. First, we compare the latency of Gyro in different configurations when the system is not overloaded.

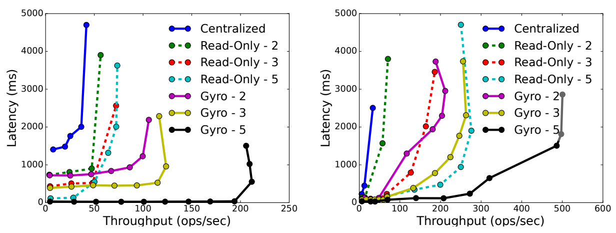
Configuration	TPC-W	RUBiS
Centralized	1390ms	416ms
Gyro – 2	671ms (2.1x)	182ms (3.3x)
Gyro – 3	436ms (3.2x)	155ms (2.7x)
Gyro – 5	29ms (47.9x)	35ms (11.9x)
Read-Only – 2	902ms (1.5x)	145ms (2.9x)
Read-Only – 3	521ms (2.7x)	131ms (3.2x)
Read-Only – 5	129ms (10.8x)	96ms (4.3x)

TABLE III: Request latency with light load in WAN.

In Table III, we report the latency improvement over the centralized setting of each configuration, from two to five servers with TPC-W and RUBiS. Gyro achieves a significant latency reduction, of more than one order of magnitude, thanks to its low need for coordination. For instance, TPC-W with 3 DBMS instances the latency is 3.2x less than that of a centralized server and 2.7x less for the read-only baseline. The performance is best when a DBMS instance is available in every geographical location of the clients. In fact, for the 5 DBMS instances configuration, the latency is 47.9x less for TPC-W and 11.9x for RUBiS. In contrast, the latency when using the read-only optimizations is only 10.8x less for TPC-W and 4.3x for RUBiS. This is because the majority of operations can be served by the local DBMS instance where clients are located. This is especially the case for the five DBMS instances case where the latency is at its lowest: 29ms for RUBiS and 25ms for TPC-W.

Latency vs. Throughput. We now shift our attention to both throughput and latency with more intense workloads (Figure 6). We stress the system by increasing the number of clients until the latency reaches 5 seconds. The single server in the centralized case starts to saturate quickly. By scaling to five locations, Gyro increases throughput by up to one order of magnitude.

Read-only optimization significantly reduces latency and increases throughput for both workloads and especially for RUBiS which is more read-dominated. Gyro, however, has a much greater impact as it allows the local execution of many more operations, both read-only and not. Overall, Gyro improves the maximum throughput compared to the read-only setting. For instance, in the five DBMS instances configuration there is an increase of the maximal throughput by 291% for TPC-W and 181% for RUBiS. In terms of scalability, Figures 6a and 6b show that Gyro scales very well until at least five geo-locations, which is a fairly high number in many practical settings. By contrast, the read-only baseline maxes out already with three DBMS instances, especially with TPC-W where the gain from using additional DBMS instances in terms of throughput is marginal due to the increased cost of coordination. The difference in terms of throughput between the WAN and LAN setting is



(a) TPC-W. (b) RUBiS.

Fig. 6: GYRO vs. baselines in a WAN setup.

due to the longer time it takes for the token to make a round. Consequently, more global operations queue up and when the token is received they take longer to process. Execution of local operations also is slowed down as they are executed concurrently with the global operations.

VIII. Conclusions

This paper introduced Gyro, a modular coordination layer that scales out unmodified DBMSs while preserving serializability in both LAN and WAN settings. We used Gyro to scale out two unmodified applications running on top of MySQL, a popular DBMS. We also described a static analysis tool to automate adding to applications the instrumentation that is necessary for using Gyro. Our evaluation shows the practical viability of using modular concurrency control: Gyro substantially outperforms a standard and popular two-phase commit protocol implementation, MySQL Cluster.

References

[1] Java parser. https://github.com/javaparser/javaparser, 2015.

[2] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: scalable sql storage for web applications. In *Procs. of SOSP*, pages 245–262. ACM, 2015.

[3] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *VLDB*, 8(3):185–196, 2014.

[4] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[5] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Procs of ECCS*, page 6. ACM, 2015.

[6] D. Barbard and H. Garcia-Molina. The demarcation protocol: A technique for maintaining linear arithmetic constraints in distributed database systems. In *Procs of Advances in Database Technology*, pages 373–388. Springer, 1992.

[7] T. Consortium. TPC benchmark-W specification v. 1.8. http://www.tpc.org/tpcw/spec/tpcw_v1.8.pdf, 2002.

[8] J. A. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *Procs. of ATC*, volume 12, 2012.

[9] S. Das, D. Agrawal, and A. El Abbadi. G-store: a scalable data store for transactional multi key access in the cloud. In *Procs. of the symposium on Cloud computing*, pages 163–174. ACM, 2010.

[10] S. Das, A. El Abbadi, and D. Agrawal. Elastras: An elastic transactional data store in the cloud. *HotCloud*, 9:131–142, 2009.

[11] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: distributed transactions with consistency, availability, and performance. In *Procs. of SOSP*, pages 54–70. ACM, 2015.

[12] C. Emmanuel and M. Julie. Rubis: Rice university bidding system. http://rubis.ow2.org/.

[13] D. G. Ferro, F. Junqueira, I. Kelly, B. Reed, and M. Yabandeh. Omid: Lock-free transactional support for distributed data stores. In *Procs. of ICDE*, pages 676–687. IEEE, 2014.

[14] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. *VLDB*, 2(1):253–264, 2009.

[15] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Procs. of OSDI*, pages 265–278, 2012.

[16] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. In *Procs. of NSDI*, pages 503–517, 2014.

[17] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *Procs. of OSDI*, pages 479–494, 2014.

[18] P. E. O'Neil. The escrow transactional method. *TODS*, 11(4):405–430, 1986.

[19] C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4):631–653, 1979.

[20] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[21] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Procs. of SIGMOD*, pages 1311–1326. ACM, 2015.

[22] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *TODS*, 20(3):325–363, 1995.

[23] C. Su, N. Crooks, C. Ding, L. Alvisi, and C. Xie. Bringing modular concurrency control to the next level. In *Procs. of SIGMOD*, pages 283–297. ACM, 2017.

[24] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Procs. of SIGMOD*, pages 1–12. ACM, 2012.

[25] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32. ACM, 2013.

[26] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *VLDB*, 10(7):781–792, 2017.

[27] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang. High-performance acid via modular concurrency control. In *Procs. SOSP*, pages 279–294. ACM, 2015.

[28] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Procs. OSDI*, pages 21–21. USENIX Association, 2000.

[29] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Procs. of SIGMOD*, pages 1629–1642. ACM, 2016.

[30] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. In *Procs. SOSP*, pages 263–278. ACM, 2015.

[31] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Procs. SOSP*, pages 276–291. ACM, 2013.

Appendix

A. Correctness

We now show the correctness of the Conveyor Belt protocol. Gyro assumes that each DBMS instance guarantees serializability for the requests it executes locally. We must show that this sequential order that is consistent with some global sequential order. A local execution order O_i for a DBMS instance i is *valid* if and only if it respects the following rules: *(R1)* A request r mapped to i is ordered according to the sequential execution order at i; *(R2)* A global request g_k mapped to $k \neq i$ is ordered according to the sequential execution order of the state update request $u(g_k)$ at i; *(R3)* A local request l_k mapped to $k \neq i$ can be ordered arbitrarily. The first rule guarantees that O_i is consistent with the execution order at instance i. The second rule ensures that state updates for remote global request must be executed in the same order as the original global requests. The third rule is acceptable since local requests mapped to remote instances do not directly influence the behavior of requests executed at instance i (and in fact, they are not executed by i).

Assume now, by contradiction, that there exists a divergence between the local (and valid) execution orders O_i and O_k at two instances i and k. Let r_i and r_k be the first two requests that diverge in O_i and O_k, respectively, starting from the beginning. Since this is the first divergence and each request appears only once in an order, r_k (resp. r_i) never appears before the divergence in O_i (resp. O_k). We assume that the divergence is unavoidable, meaning that there is no other valid ordering of O_i or O_k that avoids the inconsistency - otherwise, it would be sufficient to consider that valid alternative ordering to obtain consistency.

Consider first the case where both r_i are r_k are global requests. Since the total order of global requests enforced by the token passing scheme, this situation cannot arise and we reach a contradiction with (R2). Consider now the case where either r_i or r_k is a local request. Assume r_i is local, w.l.o.g. If r_i (resp. r_k) is not mapped to k (resp. i), it is possible to insert r_i (resp. r_k) into the position of r_k (resp. r_i) in O_k (resp. O_i) and shift all other requests to their successive position. This order would be compatible with (R3) so the divergence would be avoidable, which is a contradiction. If r_i is mapped to k and r_k is mapped to i, then we can push r_i to a later point in O_i and replace it with its following element and obtain a new order O'_i. Using the same logic with the new order O'_i we either end up in a different case, and thus reach a contradiction, or we end up in this case for all elements in the original O_i following the divergence point. In the latter case, since all elements in O_i after the divergence point are local and mapped to k, it is possible to avoid the divergence by inserting r_k into the position of r_i in the original O_i and shift all other requests to their successive position. This order would be compatible with (R1) since there are no other requests in O_i mapped to i after r_k. The divergence would thus be avoidable, a contradiction.

B. Implementation

Next, we describe how our implementation of Gyro handles the extraction of state updates that are propagated to the rest of the instances and an optimization for concurrent processing of global operations.

Extracting state updates. Gyro transparently and automatically instruments the application code to extract state updates from the execution of global client requests. It records changes to the DBMS state by intercepting interactions between the application and the DBMS, which occur through JDBC. Every time the application begins executing a global request, the instrumentation generates a request object that is used to store the state updates. Gyro then uses the request object as a wrapper to JDBC: every time the application invokes a statement s mutating the state (e.g., `UPDATE`), it does so through the requests object instead of JDBC. The request object appends s to the sequence of SQL query statements invoked within the request and then passes s to JDBC. At the end of the transaction, the sequence of SQL statements in the request object represents the sequence of state mutations that constitute the state update.

Optimizing the execution of global requests. Gyro is able to parallelize the execution of global requests (Lines 14-18 of Algorithm 1). Concurrency makes it hard to add state updates to the token in the same order in which they are executed. This is possible if the DBMS respects the following *commit ordering* property: a request t cannot invoke a commit before all requests that are serialized before t have committed. One common way for DBMSs to ensure commit ordering is to ensure serializability through the use of pessimistic locking: before a transaction accesses a row i, the transaction acquires a lock for i and releases it only after the transaction is committed or aborted. Therefore, any concurrent transaction t' for a request o' that has a conflict with t will not be able to invoke commit until t has committed and released its locks, which satisfies commit ordering.

If the DBMS supports commit ordering, Gyro activates the parallel execution of global requests. When the application requires a transaction t for request o to commit, Gyro intercepts this call, appends to a reference queue U the state update u_o produced by o, and then invokes the commit. Because of the commit ordering property, the thread executing t' will append o' to U only after t has finished appending o, so the order of the requests in U is consistent with the execution order of t and t'. Updates that do not conflict can be added to U in any order: Gyro uses a concurrent queue implementation to allow safe concurrent updates from multiple threads.

If the DBMSs does *not* satisfy the commit ordering property, Gyro simply executes requests in Q sequentially as specified by Algorithm 1