A Drop-in Middleware for Serializable DB Clustering across Geo-distributed Sites

Enrique Saurez[1], Bharath Balasubramanian[2], Richard Schlichting[3]
Brendan Tschaen[2] Shankaranarayanan Puzhavakath Narayanan,[2]
Zhe Huang[2], Umakishore Ramachandran[1]
Georgia Institute of Technology[1] AT&T Labs - Research[2] United States Naval Academy[3]
esaurez@gatech.edu, bharathb@research.att.com, schlicht@usna.edu,
bt054f@att.com, {snarayanan, zhehuang}@research.att.com, rama@gatech.edu

ABSTRACT

Many geo-distributed services at web-scale companies still rely on databases (DBs) primarily optimized for single-site performance. At AT&T this is exemplified by services in the network control plane that rely on third-party software that uses DBs like MariaDB and PostgreSQL, which do not provide strict serializability across sites without a significant performance impact. Moreover, it is often impractical for these services to re-purpose their code to use newer DBs optimized for geo-distribution. In this paper, a novel drop-in solution for DB clustering across sites called Metric is presented that can be used by services without changing a single line of code. Metric leverages the single-site performance of an existing service's DB and combines it with a cross-site clustering solution based on an entry-consistent redo log that is specifically tailored for geo-distribution. Detailed correctness arguments are presented and extensive evaluations with various benchmarks show that Metric outperforms other solutions for the access patterns in our production use-cases where service replicas access different tables on different sites. In particular, Metric achieves up to 56% less latency and 5.2x higher throughput than MariaDB and PostgreSQL clustering, and up to 90% less latency and 26x higher throughput than CockroachDB and TiDB, systems that are designed to support geo-distribution.

PVLDB Reference Format:
Enrique Saurez, Bharath Balasubramanian, Richard Schlichting, Brendan Tschaen, Shankaranarayanan Puzhavakath Narayanan, Zhe Huang, Umakishore Ramachandran. A Drop-in Middleware for Serializable DB Clustering across Geo-distributed Sites. *PVLDB*, 13(12): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/xxxxxxx.yyyyyyy

1. INTRODUCTION

Services built by AT&T and other web-scale companies such as Google and Amazon are often deployed across geo-distributed sites to satisfy the locality, availability, and performance needs of clients.[1] However, many of these services use databases (DBs) like MariaDB [54] and PostgreSQL [50] that are primarily optimized for single site deployments even when they have clustering solutions. For eample, in MariaDB Galera [28] synchronous clustering [29] all replicas are updated on each commit, which is prohibitively expensive across sites with WAN latencies on the order of hundreds of milliseconds. Similarly, in PostgreSQL master-slave [49] clustering, requests from all sites are sent to a single master replica, compromising on performance and availability.

Although new geo-distributed DBs have been developed that improve the performance of cross-site transactionality (e.g., Spanner [19], CockroachDB [17], TiDB [48]), it is often impractical to re-purpose the code of services tied to specific DBs to use these new solutions. This is especially true when the existing service involves third-party software. For example, AT&T's multi-site Service Orchestrator (SO [42]) that deploys complex virtual network functions (VNFs) relies on a third-party business process engine Camunda [8] that maintains state in MariaDB. Similarly, AT&T's Data Collection, Analytics and Events service (DCAE [21]) relies on a third-party tool called Cloudify [16] that uses PostgreSQL.[2] While middleware for DB clustering does exist, it does not provide multi-master strict serializibility [15, 34] and/or requires extensive annotation of service code [30].

In this paper, we present a novel solution called Metric that serves as a replacement for existing DB clustering solutions. The primary challenge in designing such a system is to satisfy all of the following goals simultaneously:

- Require no changes or annotations of the service code or its DB aside from turning off the latter's default clustering solution across sites; in other words, it should serve as a *drop-in solution*.
- Provide the service or user of the middleware the abstraction of a replicated multi-master DB across sites, where all replicas can concurrently process requests.
- Guarantee that all transactions are strictly serializable [47, 33].
- Build a system that outperforms a service's existing DB clustering solution, in terms of the end-to-end latency for transaction execution and throughput.

[1]A *site* is a data center at a physical location connected with other sites through a wide-area-network (WAN).
[2]Both SO and the DCAE are part of AT&T's network control plane, which is open-sourced through the Open Network Automation Platform (ONAP) effort [44].

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.yyyyyyy

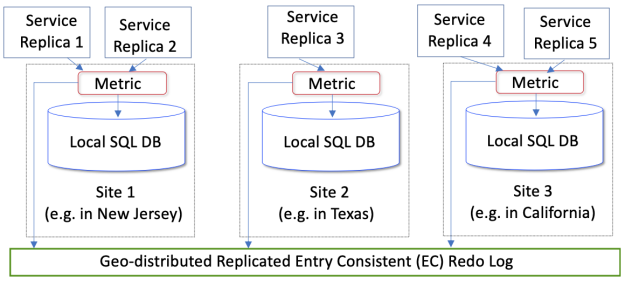


Figure 1: Overview of a Metric deployment where instances of Metric and the SQL DB are deployed on each site with a geo-distributed redo log deployed across the sites. Service replicas issue requests to the Metric process closest to them.

- Support new DBs easily with minimal additions or changes to the middleware.

Metric achieves these goals through a novel design that leverages the single-site guarantees of the service's existing DB coupled with the use of an *entry-consistent* (EC) key-value store [5, 3] to maintain a geo-distributed redo log of DB records. The EC store provides critical functionality in the form of fault-tolerant lock-based critical sections that are used by Metric to obtain table-level locks that guarantee exclusive access to the latest values of records accessed by a given transaction. Figure 1 illustrates this approach, where a Metric process executes the operations in a transaction locally on a DB, with just one round-trip per transaction across sites to commit modified records in the EC log. This is operationally not only much more efficient than both MariaDB synchronous and PostgreSQL master-slave clustering, but is also similar to optimized geo-distributed DBs [19, 17], despite the drop-in nature of the Metric solution. Further, Metric makes effective use of the EC store's higher level abstraction of critical sections that handle failures to build a redo log. The above-mentioned geo-distributed DBs design their redo log from first principles—a complex error prone process, especially considering the wider array of failures in geo-distributed systems.

A key aspect of Metric's drop-in solution is that it supports general SQL queries and parses the query to determine automatically the potentially impacted tables over which to acquire table-level locks. For our use-cases such as DCAE and SO, transactions are naturally partitioned across service replicas and have no overlap in the DB records they access, meaning that a given table is usually accessed only by processes within a specific site. For example, SO replicas typically deploy different VNFs, with each replica modifying records in distinct DB schemas. An SO replica requires access to another replica's records only when the latter fails, in order to complete VNF deployments. For this common usage pattern Metric achieves optimal performance.

Metric is implemented in Java with support for MariaDB and PostgreSQL [25]. Services use the middleware by replacing their existing JDBC driver [62] with the Metric JDBC driver for the choice of their DB. Through the use of SQL triggers and basic SQL parsing, we ensure that support for a new DB can be added with less than 1000 LOC, consisting mainly of boilerplate code for initialization, trigger management, and the mapping of DB data types to Java data types.

We evaluated Metric with strict serializability in multi-site settings across different WAN latency profiles using micro-benchmarks, use-case workloads, and TPC-C workloads. For the access patterns described above where service replicas access different tables on different sites, Metric achieves up to 56% less latency and 5.2x higher throughput than MariaDB's Galera synchronous clustering solution and PostgreSQL's master-slave clustering solution. Metric also outperforms DBs optimized for geo-distribution on the same access patterns, and achieves up to 90% less latency and 26.2x higher throughput than CockroachDB and TiDB.

We also evaluated Metric for access patterns that deviate from the expected workload, where service replicas frequently access the same tables across sites. As expected, Metric's performance drops relative to the other solutions mentioned above, which demonstrate up to 90% less latency and 32x higher throughput than Metric. We present several mitigation strategies in §9 as future work.

In summary, this paper makes these contributions:

- A novel approach to providing drop-in DB clustering across sites supported by detailed correctness arguments showing strict serializability (§3, §4).
- An implementation [25] with clustering support for MariaDB and PostgreSQL that is being deployed in production for multiple use-cases and that is open-sourced through ONAP (§5, §6).
- Experimental results validating Metric's effectiveness (§7).

A previous Metric workshop paper oriented towards edge use-cases [57] presents some of the initial design ideas related to the system. These include how the ownership API can be exposed to a client or service, and an approach for guaranteeing transactionality only to the owner of certain tables in the DB. While we retain the name for legacy reasons,[3] this paper significantly extends these concepts to encompass strict serializability guarantees, and presents complete correctness arguments, details of an implementation, and an experimental evaluation.

2. ARCHITECTURE AND OVERVIEW

Architecture. Metric provides the abstraction of a replicated geo-distributed DB that can be accessed by applications implementing higher-level services. As shown in Figure 1, each application is generally composed of multiple *service replicas* that are hosted on different sites for locality, availability, and fault-tolerance. Metric itself is also geo-distributed, with one replica per site. Service replicas submit transactions to the closest Metric process, usually at the same site and often on the same machine. A Metric process is in turn associated with an instance of a SQL DB, referred to as the Metric process's *local DB*. Currently, a given multi-site Metric deployment supports a single type of SQL DB (e.g., MariaDB, PostgreSQL), where the choice is based on application requirements. Each local DB contains at least the records accessed by transactions submitted to the Metric process at that site. The DB must support strict serializability, at least for transactions within the same node.[4] To avoid conflicts and optimize performance, any internal cross-site clustering facility provided by the DB is disabled (e.g., Galera clustering for MariaDB.) However, the DBs can use their clustering solution *within* a site as long as that solution provides strict serializability for transactions.

[3]The name is no longer considered an acronym, however.
[4]§9 describes strategies to relax this assumption

Metric's core functionality is providing efficient geo-distributed replication. This is realized using a redo log built on an underlying multi-site entry-consistent (EC) store, which provides useful consistency guarantees despite the increased latencies and failures prevalent in a WAN environment (details in §3.1). Both Metric and the EC store can be deployed and scaled as independent services across multiple geo-distributed sites.

Listing 1: Example Service Replica JDBC Code

```
1 Class.forName("com.mariaDB-metric.jdbc.Driver
    ↪ ");//register driver
2 Connection conn = DriverManager.getConnection
    ↪ (METRIC_URL,USER,PASS);//connect to a
    ↪ METRIC process (mproc) nearby
3 Statement stmt = conn.createStatement();
4 String query 1 = "select T2.id, T2.date from
    ↪ T2 inner join T3 ON T2.id=T3.id";
5 Resultset rs = stmt.executeQuery(query1);
6 String query2 = "update table T1, set T1.
    ↪ salary = 1000 where T1.id = r1;"
7 stmt.executeQuery(query2);
8 conn.commit();//commit transaction
9 stmt.close(); rs.close(); conn.close();//
    ↪ clean up
```

Overview. Listing 1 gives an example of the JDBC code that would be executed by a service replica to use Metric. Note that this is simply standard JDBC code, a key property of the solution since it allows Metric to serve as a drop-in replacement for existing DBs. The calls are intercepted by a custom Metric JDBC driver that transmits it for execution at the Metric process associated with METRIC_URL.

The service code in Listing 1 starts with a statement to import the specific Metric JDBC driver that clusters the service's existing DB, which in this example is MariaDB. The subsequent *getConnection* and *createStatement* statements involve initialization routines that create the connection between the service replica and the Metric process (*mproc*), and initialize the basic Metric data structures. Metric then performs the following sequence of actions during the course of executing a transaction, which consists of the queries followed by the *conn.commit* (details in §3.2):

1. *Initialization.* All DB schemas, tables and views are created in all of the local DBs before any query is sent to Metric.
2. *Identifying the table-set.* When *executeQuery* is invoked, *mproc* first parses the query to identify the tables impacted by the query, referred to as the query's *table-set*.
3. *Acquiring Ownership.* Using the locks provided by the EC store, *mproc* locks the relevant entries for the table-set in the redo log, which contains the latest records of all the SQL tables in a compressed format. This makes *mproc* the *owner* of these tables, with exclusive read and write access. Note that while it would potentially be useful to support record-level locking, it is difficult to identify the precise records that are modified when parsing the query in step 2 given the inherent complexity of such queries.
4. *Populating the Local DB. mproc* reads the redo log and populates the local DB with records of the table-set. The EC store semantics guarantee that the owner of a table-set will read the latest values of the records in the table-set from the redo log.
5. *Executing the Query on the Local DB. mproc* then executes the query on the local DB and uses DB write triggers to populate a private Metric table in the DB called the *local tx-queue* table with the modified records.
6. *Committing the Transaction.* On executing *conn.commit*, *mproc* first commits the transaction in the local DB, then updates the redo log at a quorum of replicas with the new values of the records modified by this transaction tracked in the *local tx-queue table*, and finally clears the records from the local tx-queue table. This is the commit point, meaning that the changes will be rolled forward even if *mproc* subsequently fails.

3. Metric SOLUTION

This section provides a detailed description of the Metric solution, starting with the design of the geo-distributed EC redo log and followed by how Metric uses this redo log to execute transactions. Finally, we describe failure handling, an aspect that is especially important for geo-distributed deployments with complex failure modes.

3.1 EC Redo Log

Background. Entry consistency as originally defined for shared memory systems specifies that data shared among multiple processors becomes sequentially consistent at a processor only when it acquires a synchronization object that guards the data [5]. In prior work done as part of ONAP [3, 40], we built a replicated geo-distributed key-value store called MUSIC in which EC semantics were expanded and re-purposed for multi-site settings and their more complex failure modes. Key properties of MUSIC were formally verified during the development process, and the system has been used in production since 2017. Since Metric is designed to work with any key-value store that provides EC guarantees and not just MUSIC, we henceforth refer to the underlying store generically.

The EC store provides traditional key-value operations augmented with provisions for locking. In particular, to modify the value of a set of keys, a client first acquires a unique lock to the keys through an *acquireLock* function. On acquiring the lock, the lock-holder can read the guaranteed latest values of the keys using the *critical get* function, and can perform exclusive, sequentially consistent updates to the keys as part of a critical section using the *critical put* function. Non lock-holders attempting these critical operations are rejected by the EC store. While lock acquisition requires distributed consensus [35, 43, 55] across the geo-distributed replicas of the EC store, the critical reads and writes to a key's value use relatively efficient quorum operations. The EC store uses heartbeats to detect lock-holder failures and release locks that they hold. For retrieving values without locking, the EC store also provides *getOne* and *getQuorum* functions that return the value of a key from any single replica or from a quorum of replicas, respectively. However, the value returned is not guaranteed to be the latest value since no lock is used.

EC Store Guarantees. The EC store provides the following guarantees that are critical for realizing the redo log:

- There can be only one lock holder for any key; this lock holder has exclusive read/write access to that key and always reads the latest value of that key.

Time-based UUID (key)	Table-set (value)	Tx-queue (value)
UUID3	T1, T2	(T1.R1, T2.R3, T1.R4) (T2.R5, T1.R1) (T1.R8)
UUID2	T2, T4	(T2.R9, T4.R1) (T2.R6) (T4.R3)
UUID1	T3	()

Figure 2: An example of the EC Redo Log table built on the EC key-value store.

- If a lock holder to a key fails, the EC store preempts the lock holder by forcibly releasing the lock to the key. Even if the failure happens while the lock holder was writing to the key, the EC store ensures that the next lock holder reads the latest value of that key.
- If a prior lock-holder that was incorrectly detected as failed and preempted due to imperfect failure detection attempts to read or write to any key, the EC store rejects these operations.

Redo Log. The EC store is used to construct a redo log that enables Metric processes to acquire ownership of DB tables, to perform exclusive reads and writes to those tables, and to maintain a history of modifications. An entry is appended to the redo log either when a Metric process initially creates a table in the local DB or when it needs to acquire ownership of tables during query execution.

Figure 2 gives an example redo log, where each row is a key-value pair representing a single log entry. Each entry has the following contents:

- A unique time-based UUID that serves as the EC store primary key for that row. These UUIDs are Cassandra's `timeuuids` [11]—standard SQL UUIDs [61] augmented with the time of the UUID's generation—inherited from our use of Cassandra to implement the redo log (see §6). While the entries are shown here sorted in descending order by UUID for convenience, since Cassandra does not allow sorting on just primary keys [10], these entries are sorted only when read from the redo log.
- A *table-set* containing the DB tables accessed by the query that appended the entry. If the entry has been locked by using *acquireLock* with the associated UUID, the executing Metric process is the current owner of the tables in the table-set. Note that any given table may also appear in an entry with a smaller UUID, indicating that ownership of that table has transitioned at some point to the current owner.
- A *transaction queue* (tx-queue) containing the records modified by queries executed by the owner of the tables in the entry. The queue is maintained in compressed form, and records are added to the queue in order of execution. The tx-queue starts out empty when an entry is created and grows as records are modified. The collection of tx-queues across the entire redo log provides a historical trace of updates to the records.

Figure 2 illustrates the contents of a redo log at a certain snapshot of time. Assume that the entry with key UUID3 is locked by some Metric process *mproc*, meaning that *mproc* is the current owner of DB tables T1 and T2. In the tx-queues, Ti.Rj indicates that record Rj of table Ti has been modified, and the records within each parenthetical clause correspond to a single transaction. In the figure, the tx-queue grows from left to right. Hence, the tx-queue for the entry UUID3 indicates that after acquiring the lock, *mproc* committed three transactions to tables T1 and T2, where the first transaction modified R1 in T1 followed by R3 in T2 and finally R4 in T1. Since each table can be owned by at most one process, this implies that UUID2 is not locked since its table-set also contains T2; this indicates that ownership of T2 transitioned at some point to *mproc*. The entry with key UUID1 containing table T3 has no tx-queue, indicating that the process that has locked UUID1 and owns T3 is in the midst of making updates, or that no updates were done.

The EC store's higher-level abstraction of critical sections and the associated guarantees make it relatively easy to build a redo log, handle failures, and reason about correctness. While DBs like Spanner and CockroachDB also rely on a geo-distributed redo/undo log, they build their solutions from scratch by combining distributed consensus with shared data structures to handle failures rather than building on an underlying system that effectively abstracts away failures by providing similar assurances.

3.2 Executing Transactions

This section elaborates on the steps for executing a transaction outlined above in §2 by describing how queries are executed by the Metric process *mproc* to which they are submitted. In §4, we show how this solution and EC redo log guarantees together ensure that transactions are strictly serializable across different DB/Metric processes from the perspective of the service submitting the transactions.

Initialization. As part of the initialization of Metric, we assume that all DB schemas, tables, and views are created in all of the local DBs before any transactions are executed. Further, whenever a view is created, we create a *view cache* at each Metric process that contains the mapping between the view name and the set of tables that will be queried in the view. As mentioned in §3.1, Metric also creates corresponding entries in the redo log for each table. Since our use-cases do not demand it, we currently do not support dynamic modification of table definitions to add or delete columns, views, or schemas. Such modifications need to be performed at each of the local DBs offline when no client requests are being serviced by Metric. As part of this process, the view caches at all Metric processes are also cleared.

Identifying the table-set. Upon receiving a query in a transaction, *mproc* first identifies the query's *target table-set*. This set consists of: (1) all tables whose records may be modified by the query including insertion/deletion, and (2) all tables that have a foreign-key relationship with any table in (1), i.e., tables that have a column that is a primary key for a table in (1). Despite the complexity of SQL queries, it is easy to identify all the tables that are affected by a query just by basic text parsing (e.g., T2, T3 in query 1 and T1 in query 2 of Listing 1). For (2), *mproc* obtains primary/foreign key information by reading the local DB's system tables. To account for views, on extracting a table name from a query, we perform a look-up in *mproc*'s view cache to check if it is a view and identify the precise tables being modified. Note that our current implementation does not include views, although we are in the process of adding that functionality to support production deployments.

Acquiring Ownership. After identifying a query's target table-set, *mproc* needs to obtain ownership of these ta-

bles to ensure that it has the latest values and to guarantee exclusive read/write access. This is done using the locking abstractions provided by the underlying EC store, as shown in Listing 2. For simplicity, we omit the code for handling failures from this listing; these considerations are addressed in detail in §3.3.

Listing 2: Pseudocode for Acquiring Ownership

```
own(table-set target) executed at mproc:
  if (mproc does not own all tables in target
    ↪ )
    release all the locks held by mproc;
  create new entry E-new in redo log;
  while (acquireLock (UUID of E-new)) skip;
  for each table T in target do:
      E = findLatest (T); //function
    ↪ described in the text
      while (acquireLock (UUID of E)) skip;
      if (findLatest (T)) != E) //another
    ↪ contending process has created an
    ↪ entry
        return nack;
      else
         add T to table-set of E-new; //mproc
    ↪  is now the owner of T
  return SUCCESS;
```

The first step is for *mproc* to determine if it already owns all the tables in the target table-set. This will be true if it retained the locks for redo log entries that have at least these tables in their own table-sets from, for example, a previous transaction. In this case, *mproc* can proceed with the next step of populating the local DB. However, if *mproc* does not already own all the tables, it relinquishes ownership of the tables it does own by releasing the locks to those entries. It then creates a new entry E-new with an empty table-set, and obtains the lock to that entry. Assuming *mproc* successfully executes *own*, the table-set of E-new will contain all the tables in *target*. This ensures that all tables owned by a process are present in the table-set of one consolidated redo log entry where the *tx-queue* can be updated atomically.

At this point, *mproc* must obtain ownership of each table T in its target table-set by locking the most recent redo log entry E containing T in its table-set. To obtain this entry E, Metric uses a simple helper function *findLatest(T)*, which iterates over each entry in the redo log using the *getQuorum* function (which does not require a lock), and then sorts them according to UUID and returns the latest entry that contains T in its table-set.[5] However, there are scenarios where the latest entry containing T may be missed if *mproc* had been in contention for the lock to E with another process. Specifically, *mproc* has to check for the latest entry a second time using *findLatest(T)* in case another process (a) had held the lock for E, and (b) released it after creating a new entry that has T in its table-set. In this scenario, *mproc* is guaranteed to find the new entry on the second attempt. Finding a newer entry implies that *mproc* does not have the most recent version of T, so it returns a nack indicating that ownership cannot be acquired at this point. When this happens, *mproc* releases all the locks it acquired during the execution of this algorithm (code omitted from Listing 2 for simplicity). The technical report [39] describes an optimization where tokens are used to identify the latest entry in the redo log that contains a given table, resulting in a more efficient implementation of *findLatest(T)*.

As an example of ownership acquisition, consider starting with the redo log in Figure 2. To acquire ownership of T2 in the target table-set (T2, T3) of query 1, *mproc* first creates a new locked entry in the redo log with the key UUID4, acquires the lock to the entry with key UUID3 since it is the latest entry containing T2, and then updates the table-set of UUID4 with T2. Following similar steps for T3 results in (T2, T3) as the table set for UUID4. Similarly, execution of query2 causes a new entry with UUID5 as the key to be created with table-set (T1). At this point, *mproc* holds the locks for both UUID5 and UUID4, and owns tables T1, T2, and T3.

Our algorithm creates a new entry with the required table-set, but it would be feasible to re-use entries in the redo log that already contain the required tables. Our choice to create these new entries was driven by a desire to facilitate concurrency across sites. In our running example, since *mproc* has new locked entries with T1, T2 and T3 and has released locks to the older entries containing these tables, another Metric process is free to acquire the lock to the entry with key UUID2 if it wants ownership of T4.

Populating the Local DB. Before executing a query on the local DB, *mproc* has to ensure that the local DB has the most recent records for each table T in Listing 2 for which it had to acquire ownership, i.e., the latest values of T's records that are present in the tx-queues of a quorum of redo log replicas. To do this, *mproc* relies on the fact that acquiring ownership of T implies that the redo log is guaranteed to have the latest values of all entries that contain T in the table-set (see §4 for correctness arguments). *mproc* reads the tx-queues of all such entries, sorts them in increasing UUID order, identifies the records in T that were modified by past transactions, and applies these changes to the local DB. For example, after acquiring ownership of T2 and T3 for query 1 in Listing 1, *mproc* iterates through the redo log and applies the updates to T2 in the local DB for records R9 and R6 (from entry with key UUID2) and then for R3 and R5 (from entry with key UUID3); no updates are needed to T3 since the redo log shows no history for the tx-queue of T3. The technical report [39] describes optimizations that use a background thread to read the redo log through a series of table-specific pointers to populate the local DB.

Executing the Query on the Local DB. After ensuring that the local DB has the latest records of all the relevant tables, *mproc* now executes the query on the local DB. While read queries simply involve reading the local DB, write queries are more complex since the records they update need to be captured. This is done using the triggers that most DBs like MariaDB and PostgreSQL provide to identify the precise rows updated by the write query. Specifically, whenever a row is inserted, updated, or deleted, a trigger fires that calls a custom Metric function with the old and new values of that row. This function populates a private Metric table *local tx-queue* in the local DB with the row that was modified. An entry in this tx-queue is similar to an entry in the redo log tx-queue shown in Figure 2.

Committing the Transaction. On receiving *conn.commit*, *mproc* first tries to commit the transaction in the serializable local DB, which will execute the commit only if there are no other conflicting transactions within that

[5] §4 shows why using just the quorum operation without any locking is sufficient for correctness.

DB. Then *mproc* uses critical operations to write the entries from the local tx-queue to the relevant tx-queue in the redo log, thereby ensuring that the records modified by the transaction are geo-replicated. For the above example, this involves updating the tx-queue in the entry with key UUID4 that was newly created for query 1. This is the commit point, meaning that even if *mproc* subsequently fails, other Metric processes have access to these records in the redo log. Finally *mproc* clears the committed entries from the local tx-queue. If any of these operations fails, *mproc* returns failure to the service that submitted the transaction.

Releasing Ownership. At this point *mproc* owns all the tables potentially impacted by the transaction. The policy of when this ownership should be relinquished by releasing the locks to redo log entries can be configured by the service to balance the trade-off between the cost of acquiring ownership and enhancing concurrency across sites. Specifically, the service can configure the number of transactions α after which the Metric process will release ownership. For example, if a service has transactions to the same tables underway at different sites, it may be prudent to release ownership at the end of each transaction or after a small number of transactions so that another Metric process can acquire ownership quickly. On the other hand, for our use-cases the Metric process never releases ownership voluntarily since ownership transition is only needed for initialization and during certain failure scenarios (details in §3.3).

For simplicity, this section assumes single-threaded Metric processes. The technical report [39] describes how Metric supports multi-threading using just process-level locks and standard *wait-notify* blocks.

Solution cost. Transactions received at a Metric process are executed efficiently at the local DB with cross-site quorum updates for geo-replication of state during commits. This is not only much more efficient than existing Galera synchronous and PostgreSQL master-slave clustering, it is operationally similar to optimized geo-distributed DBs [19, 17], despite the drop-in nature of the Metric solution. The most expensive parts of the Metric solution are acquiring ownership, which requires cross-site consensus, and populating the local DB, which requires cross-site quorum reads. However, our optimized implementation ensures that these operations only take ∼460 msecs and ∼3 secs respectively (§7). This is acceptable for our use-cases, especially since ownership over a set of tables is typically acquired only on initialization and failovers (§5), and is retained for multiple transactions. In §9, we discuss several ways to reduce this overhead as part of future work.

3.3 Handling Failures

Our system model assumes distributed nodes that communicate using messages that can be lost or re-ordered. To overcome the impossibility of distributed consensus in asynchronous systems [26], we assume partial synchrony [23, 24] where there are sufficient periods of communication synchrony with an upper bound on message delay. Nodes can suffer crash failures [36], which implies that other nodes cannot distinguish between a failed node and one that is slow to respond and/or unable to communicate. This is relatively common in geo-distributed systems where link failures [63, 51] can partition a node from some subset of other nodes.

One failure scenario occurs when a service replica fails to obtain a response from the Metric process to which it is connected within a specified timeout interval or receives a nack. The return of a nack indicates that the Metric process is alive, but a) is unable to connect to or receives an error response from either the local DB or the redo log, or b) is unable to obtain locks to execute transactions since other processes are holding them. If either Metric times out or a nack is received, the service replica considers the Metric process to have failed and treats it the same way it would a failed DB node. For example, it can retry the entire transaction at a different Metric process. The Metric process that subsequently executes this transaction will perform all the steps described in §3.2, i.e., obtain exclusive ownership to the necessary tables, populate its local DB, execute the queries in the transaction, and then commit it.

A Metric process can acquire locks and execute critical EC store operations on an entry if it can reach a quorum of non-failed EC store replicas, while it can execute non-critical operations that do not require a lock even if it can just reach one non-failed replica. If a Metric process fails to obtain a response from the EC store replicas within a timeout period or receives a nack, it has to retry the function—usually at a different EC store replicas—until the operation succeeds, the Metric process fails, or the Metric process is told it is no longer the lockholder. If the Metric process does not receive a response after these retries, it must not attempt any other EC store operation on the key; in this case, the Metric process can simply exit the code and attempt to modify the value of the key in a new critical section if desired. All locks are associated with a configurable lease period, and the EC store will forcibly release the lock from Metric processes once the lease period expires.

We use Nagios [41] to detect failures in the local DBs, EC store replicas, and Metric processes, and to restart them. We assume that there are enough replicas of each component and that failures are infrequent enough that all proper requests eventually succeed.

4. STRICT SERIALIZABILITY

The abstraction that Metric offers to services is a replicated database, which we refer to as the *Metric DB*. In the following paragraphs we show how the Metric solution described in §3 combined with the EC Store Guarantees described in §3.1 together ensure strict serializability. Specifically, we show that (1) the solution ensures that at most one Metric process can execute transactions on any table in the Metric DB at any given time, and (2) that this process always reads and writes the table's latest values. Due to space constraints, we defer the extended definition of serializability based on [33] and the proofs of theorems to [39].

To perform transactions on tables, a Metric process has to first acquire ownership of the tables. We define the notion of ownership formally in the following definition.

Definition 1. *The* latest *redo log entry is defined as the entry with the highest UUID present in at least a quorum of redo log replicas.*

Definition 2. *The Metric process that holds the lock to the latest redo log entry containing T is defined as the* owner *of T, where a redo log entry* contains *T if T is member of the entry's table-set.*

Observation 1. *Since the latest redo log entry containing T is uniquely defined, there can be at most one owner for any table T.*

For example, in Figure 2, while the redo log entries with keys UUID2 and UUID3 both contain T2, only the Metric process that holds the lock to UUID3 is the unique owner of T2.

We now define the guarantee provided by the *own* function when executed by a Metric process *mproc*.

Guarantee 1. *On successfully executing the* own *function in Listing 2,* mproc *becomes the unique owner of the tables in* target*.*

In [39], we show that successfully executing *own* guarantees that *mproc* is the lock-holder for the latest redo log entry containing T for each table T in target. The two calls to *findLatest(T)* and the locking semantics of the EC store ensure that even if multiple processes are trying to lock the latest entry, only one of them will succeed.

We now move on to the critical Ownership property.

Ownership Property. *Only the unique owner of a set of tables can commit a transaction that modifies records in those tables.*

The proof is based on the fact that, to commit a transaction, the owner of a set of tables updates the tx-queues of all the latest redo log entries containing the tables. By definition, this owner holds the lock to all these entries, meaning that the EC store will accept the modification to the tx-queues. Further, the EC store detects failures in lock-holders and forcibly releases their locks. Hence, the reads and writes of a prior owner of a table that was preempted will be rejected by the EC store.

We have demonstrated so far that that a Metric process has exclusive access to a set of tables once it acquires ownership to those tables. Next, we show how the owner also has access to the latest value of these tables.

Definition 3. *The* latest value *of a record in the Metric DB is the value of the most recent write to this record performed in a successful transaction by a service replica.*

Invariant 1. *For each table T that is currently owned by some Metric process, the latest value of each record R in T is the top-most value of R in the tx-queue of the latest redo log entry that contains R.*

We prove this invariant in [39] using an inductive approach referring to the variables used in Listing 2. Before returning success to the service replica, *mproc* first appends the records modified in each transaction to the tx-queue of E-new using a EC store critical operation that modifies a quorum of redo log replicas. Hence, the invariant holds for every record modified by the current owner of a table *mproc* in a successful transaction. Even for the subtle case when the previous owner of E fails while updating the tx-queue and the EC store forcibly releases the lock to E, the EC store ensures that *mproc* is guaranteed to read the latest state of E once it acquires the lock, despite the forcible release. Based on this invariant, we prove the Latest-Record property.

Latest-Record Property. *The owner of a table always reads the latest value of any record in the Metric DB.*

When a Metric process acquires ownership of a table T, it populates the records of T in its local DB with the values of the records in the redo log. Invariant 1 ensures that after this step, the local DB has the latest values of the records of T and all other tables owned by *mproc*. The strict serializability guarantees of the local DB ensure that the owner of a table continues reading the latest values of all the table's records as it commits new transactions.

The Ownership Property guarantees that transactions that affect at least one table in common are executed one at a time by the owners of the tables affected by the transaction. The Latest-Record Property guarantees that the values committed in a transaction are read by all subsequent transactions despite ownership transitions. Hence, the sequential history of the values of modified records match the exact order in which they are modified by successful transactions, thereby guaranteeing strict serializability.

5. PRODUCTION USE-CASES

This section describes how Metric supports multi-site DB clustering in AT&T's network control plane. Solutions for these use-cases are now being implemented and will be deployed in production in 2021.

Service Orchestrator (SO) is a core service in AT&T's network control plane responsible for the instantiation, release, migration and relocation of VNFs on various AT&T sites [42]. SO executes well-defined BPMN (Business Process Model and Notation [6]) process workflows to complete its objectives and is typically triggered by the receipt of VNF instantiation requests by the client. The key aspect of SO relevant to this paper is that SO's BPMN flows are specified and executed in an internal BPMN engine called Camunda [4], which maintains all the state associated with these workflows in MariaDB. SO is designed to operate in *multi-master mode* across sites, where any SO replica can process instantiation requests. If a SO replica (its Camunda engine) is unable to communicate with a certain DB replica due to failure, it is designed to seamlessly failover to another DB replica and continue operations as long as the replicated DB provides transactional guarantees.

SO is currently deployed in production, but only within a single site where each SO replica communicates with a shared MariaDB-Galera cluster. However, production requirements dictate that it be deployed across multiple geo-distributed sites, something that is prohibitively expensive using MariaDB-Galera as mentioned in §1. This solution is especially suboptimal in this context since the latency perceived by the sequential Camunda calls to the DB translates to long delays in service deployment and instantiation.

This problem can be addressed with a much more efficient Metric-based solution identical to Figure 1, where the service replicas are SO replicas and the SQL DB is MariaDB with its cross-site Galera clustering turned off. SO replicas across different sites can concurrently process instantiation requests by communicating with the Metric process of its choice, typically the one closest to the SO replica. For a specific VNF instantiation request, an SO replica sends all DB accesses to the same Metric process. Every SO replica has its own DB schema and all Camunda SQL queries it generates are specific to the tables in this schema. As part of initialization, all distinct schemas and the tables within them are created in the local DB at each site.

As it executes transactions, a Metric process acquires ownership of tables. The set of these tables is guaranteed to be disjoint from the set of tables acquired by every other Metric process since each Metric process is executing requests from different SO replicas, each with their own distinct schema.

Hence, during failure-free operation there is no ownership transition across Metric processes. Only on detection (or suspicion) of failure does an SO replica redirect requests to another Metric process, which will acquire ownership of the tables of the failed Metric process as and when required. Since failures are relatively infrequent, this usage pattern ensures high performance, as quantified by experiments in §3 that compare the performance of Camunda benchmarks [9] across different clustering solutions.

The **Data Collection, Analytics and Events (DCAE)** service [21] in AT&T's network control plane executes independent closed loop flows that detect anomalous events about VNFs and triggers appropriate actions from other network controllers. For example, a closed loop flow can detect hot spots in the physical hosts on which a VNF has been deployed and trigger SO to perform migrations of VMs belonging to that VNF to lesser congested hosts. These closed loop flows are specified and executed by DCAE using a tool called Cloudify [16] that uses a PostgreSQL DB to maintain state. While there is a production requirement to deploy DCAE replicas across geo-distributed sites in multi-master mode, as mentioned in §1 there are problems associated with transactional clustering of PostgreSQL across sites. This can be addressed using a Metric solution for PostgreSQL clustering similar to the one described above for SO. Since again there are no overlapping transactions across independent DCAE flows executing in different DB schemas, the usage pattern of DCAE will also achieve high performance with Metric.

This design pattern of multi-master replication with each replica executing independent flows is also common across other geo-distributed services in AT&T's network plane and elsewhere. For example, we are currently designing a Metric-based solution for AT&T's SDN Controller (SDNC) [59]. This controller is based on OpenDaylight [52], which maintains state in MariaDB or PostgreSQL in a manner similar to the other services described above.

6. IMPLEMENTATION

This section provides an overview of how the Metric solution and optimizations described in [39] have been implemented. The code [25] is open-sourced as part of ONAP.

Metric has been implemented in ∼9000 lines of Java code. We use Avatica [1], a subproject of Apache Calcite [7], to implement the Metric JDBC solution. Specifically, Avatica provides a framework for JDBC API management that can be used to build JDBC-compliant custom solutions like Metric. As shown in Figure 3, the service replica simply imports the Avatica client driver with no change. The driver code sends every JDBC request from the service code to the Avatica server process which resides at the Metric process. Communication between the Avatica client and server is serialized using Protobuf [53]. We realize the Metric solution by implementing certain functions in the JDBC server process, including the *getConnection*, *createStatement*, *executeQuery* and *commit* functions described in §2. Metric also leverages Apache Calcite for parsing the SQL queries and extracting the target table-set.

Our EC store implementation is ONAP's MUSIC v3.2.1 [40], which has been deployed and used in AT&T's production deployments since 2017. MUSIC is built as a Java library that interacts with a multi-site Cassandra 3.11.4 deployment. The library contains all the APIs and code for

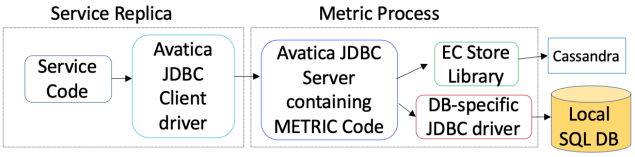


Figure 3: Components in the Metric Implementation.

the locks and key-value access necessary for entry consistency. Hence, as shown in Figure 3, each Metric process includes the EC store/MUSIC library and uses the MUSIC EC store APIs to construct the redo log. Since Metric writes a potentially large tx-queue on each commit, we maintain it in the local DB as a Protobuf object and compress it using Zlib [67] before writing to the redo log.

In Metric, the only requirements for the local DB are that it provides strict single node serializability and supports DB triggers on writes. Hence, configuring a service to use a different DB just requires importing a different Metric JDBC driver; the code for the Metric process itself and the redo log remain the same. For example, to provide clustering for MariaDB the Metric process imports the MariaDB JDBC driver [38]. Metric currently supports clustering for PostgreSQL and MariaDB, packaged as two distinct solutions that services can use based on their requirements. These solutions differ by just ∼1000 lines of boilerplate code for initialization, trigger management, and the mapping of DB data types to Java data types. For example, the Java float data type maps to the MariaDB float data type but maps to the PostgreSQL real data type. Hence, as described in §3.2, it is very easy to add support for new DBs.

7. EVALUATION

This section demonstrates that for the access patterns in our use-cases, Metric outperforms the clustering solutions of both MariaDB and PostgreSQL across geo-distributed sites for latency microbenchmarks, use-case workloads based on Camunda benchmarks, and TPC-C workloads. It also outperforms CockroachDB and TiDB, which have been optimized for geo-distribution. While these solutions outperform Metric for other access patterns as expected, we offer several mitigation strategies in §9.

7.1 Methodology

Setup. All our experiments are performed on three virtual machines (VMs), each of which logically represents a geo-distributed site. The VMs are hosted using KVM on a 2xAMD EPYC 7501 32-Core Processor server, with 256 GB of RAM and a 4x240 GB SATA SSD. We use Ubuntu 18.04 as both guest and host OS. To capture the effect of geo-distribution, we emulate WAN latencies across the VMs/sites using NetEm [32].[6] We use two latency profiles, one for sites located across the U.S. and Europe (lUsEu) and one for sites located within the U.S. (lUs). For lUs, the three sites, **S1** (N. Cali.), **S2** (Ohio), and **S3** (Oregon) have round trip latencies of 53.8, 24.2, and 72.1 ms for S1-S2, S1-S3, and S2-S3 respectively, based on EC2 measurements [2]. These values are independent of which of the two sites is used as the starting point. We defer results for lUsEu to [39] since the trends are similar to lUs. These profiles reflect the target production deployments of AT&T and other carriers involved in ONAP.

[6]We did not perform these experiments on Amazon's EC2 cloud to have a tighter control over traffic and latencies.

Comparisons. In our experiments, we focus on the following broad comparisons. The first is the Metric implementation described in §6 with clustering for MariaDB 10.3.18 (*M-Metric*) against MariaDB 10.3.18 with Galera synchronous [29] clustering (*GaleraSync*), which provides multi-master serializability across sites. The second is Metric with clustering for PostgreSQL 11.5 (*P-Metric*) against PostgreSQL master-slave [49] (*PostgresMS*) or active-passive clustering where the single master deployed on Site 1 processes all requests and replicates them synchronously across a quorum of sites.[7] Finally, we compare M-Metric and P-Metric against CockroachDB 19.2 and TiDB 3.1.1, both of which are highly optimized to provide multi-master serializability across sites. We carefully configured CockroachDB [14, 13] to use the "Follow-the-Workload" strategy [12], since that is the only one that provides serializability for both reads and writes, tolerates site failures, and does not require any locality annotations to the SQL tables. We configured TiDB using the specifications in [64, 65], and followed their recommendation to use three additional nodes to host the underlying TiDB key-value store.

Due to the different pair-wise latency between the sites and the asymmetric PostgresMS deployment (one master, two slaves), in our comparisons, we highlight the site-specific throughput and latency measurements.

Workloads. For the latency microbenchmarks we use OpenJDK JMH [45] as the framework for sending transaction requests and measuring the latency on all systems. For the TPC-C [31] benchmarks, we use the OLTP-Bench [22] framework. For the use-case benchmarks, we use the Camunda performance test suite [9]. Unless specified otherwise, to follow the usage patterns described in §5, we maintain different tables across DB/Metric instances on different sites, and hence, issue non-overlapping transactions, i.e., transactions that modify different tables. For this usage pattern, we acquire ownership of the relevant tables during initialization at each Metric process and no locks are obtained during execution of the queries.

Deployment. In all our comparisons we deploy a replicated cluster of each of the above-mentioned systems with one instance of each DB on each site. Depending on the experiment, a JMH, OLTP-Bench or Camunda process (i.e., the service replica) co-located on each site/VM sends transactions to the systems. While for the Metric and GaleraSync solutions the requests are sent to the Metric process or DB co-located on the same site, all requests are sent to Site 1 for the single-master PostgresMS solution.

7.2 Latency Microbenchmarks

We evaluate the end-to-end latency as perceived by the services/users of the five systems described above. Specifically, we measure the difference between the beginning of a transaction at a service and the time at which it receives an acknowledgment for its commit. With respect to Listing 1, this corresponds to the difference between the time at which *conn.commit* returns and the time at which the first query was sent to the DBs. For our latency measurement using JMH, we create one table in the DB, populate it with 50 rows each of size 46 bytes, and perform two experiments each with one transaction updating either 10 or 50 rows respectively. In the update, the contents of a string column are replaced and a counter is incremented in each row. Latency measurements for reads are deferred to [39] since read-only transactions have no impact on geo-distribution in any of these solutions. We first run five warm-up iterations followed by 20 measured iterations. Each of these iterations lasts 10 seconds during which as many operations as possible are executed.

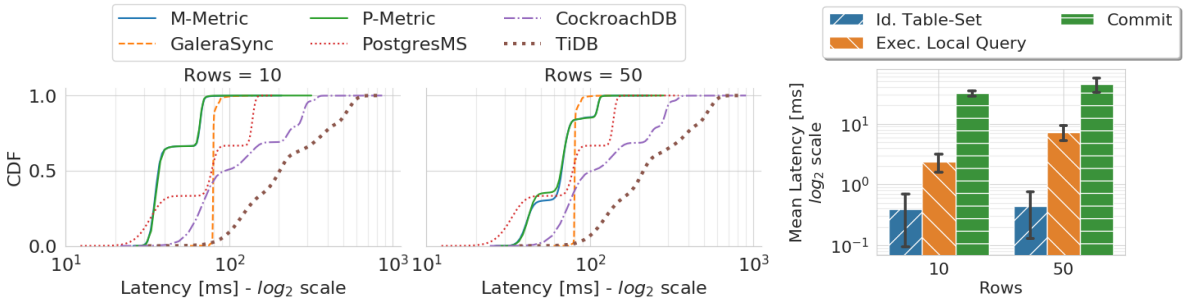
Overview. Figure 4(a) shows the latency CDF for the various solutions with different number of rows updated in each transaction (10 and 50) using a *log* scale x-axis. Figure 4(b) shows the corresponding operation-level breakdown of the mean latency with standard deviation bars for the individual operations in M-Metric (breakdown was similar for P-Metric). The CDF has steep latency steps for the Metric systems and CockroachDB due to the asymmetry in the pair-wise latencies among sites and also because these systems wait for replies from a quorum of replicas. So two-thirds of the total requests (66th percentile)—i.e, those sent from the JHM thread on Site 1 or Site 2—experience far less latency than the ones sent from the thread on Site 3. For PostgresMS, which also waits for a quorum of replies, the requests sent from the thread on Site 1 that runs the master experience less latency compared with the latency of requests sent from the thread on Site 2 to the master on Site 1. The maximum latency is experienced by the thread sending requests from Site 3 to the master on Site 1. As expected, the CDF for GaleraSync does not have any steps since it waits for replies from *all* replicas.

M-Metric and GaleraSync. For the 33th percentile latency, M-Metric has 56% and 37.3% less latency than GaleraSync for 10 rows and 50 rows, respectively. For the median or 50th percentile latency, M-Metric has 54.5% and 16% less latency than GaleraSync for 10 rows and 50 rows, respectively. M-Metric clearly outperforms GaleraSync, perhaps because of the latter's need to wait for replies from all replicas as opposed to M-Metric, which only waits for a quorum of replies.[8] For the 90th percentile latency, while M-Metric has 17.6% less latency than GaleraSync for 10 rows, the latter has 25.9% less latency than M-Metric for 50 rows. This is because of the relatively larger tx-queue with an increased number of rows that need to be written to the redo log. While the number of rows in a transaction is typically less than 50 for our use-cases, we plan to investigate ways to mitigate this cost as part of our future work.

P-Metric and PostgresMS. For the median or 50th percentile latency, P-Metric has 53% and 14.1% less latency than PostgresMS for 10 rows and 50 rows, respectively. For the 90th percentile latency, P-Metric has 51.4% and 19.8% less latency than PostgresMS for 10 rows and 50 rows, respectively. Hence, while the PostgresMS clustering protocol is quorum-based like P-Metric, the time taken to route requests across the WAN to the master in PostgresMS imposes considerable overhead, resulting in P-Metric's superior performance. For the 33th percentile latency, PostgresMS has 3.7% and 22.7% less latency than P-Metric for 10 and 50 rows, respectively. These points in the CDF correspond to the requests sent directly from the JMH thread located at Site 1 to the PostgresMS master without any routing across

[7]To the best of our knowledge there is no official PostgreSQL solution that provides multi-master serializability across sites.

[8]There maybe other reasons that account for this difference, but they are difficult to ascertain from our experimental study of GaleraSync or from documentation.



(a) Latency CDF across Sites **(b)** M-Metric Latency Breakdown

Figure 4: Latency Microbenchmarks (P- and M-Metric are indistinguishable in (a))

the WAN. Even though P-Metric uses PostgresMS as its local DB, it incurs additional but constant overhead for maintaining a separate tx-queue in the local DB and for fetching and serializing its contents before writing to the redo log. PostgresMS, which is not a drop-in solution, does not incur this overhead since it can read directly from its internal write-ahead log and send it to the replicas.

P/M-Metric and TiDB/CockroachDB. Both of the Metric solutions outperform TiDB and CockroachDB across all the data points, with 54-89% and 14-80% less latency, respectively. In TiDB, this follows because the owners of key-ranges are spread across sites, which can lead to extra cross-site accesses for writes in addition to the transaction commit. Also, TiDB needs to allocate a globally unique id per transaction using a centralized service. This is is contrast with Metric, where all writes are performed locally with only commits going across sites. For CockroachDB, Metric has consistently lower latency, even though the CockroachDB results used in the graph are from only those runs where writes were served by lease-holders/owners on the same site, the best case from a latency perspective. While the reasons for Metric's lower latency are not entirely clear, we speculate that this is because the quorum write operation used for committing a transaction in Metric is more efficient in practice than the distributed consensus used for writes in CockroachDB.

Latency Breakdown. In the latency breakdown in Figure 4(b), the labels in the figure are abbreviated versions of the labels used in the bullets in the overview of §2. While identifying the table set and executing the query on the local DB has a mean latency of just 0.4-0.4 msecs and 2.38-7.28 msecs respectively, committing the transaction has mean latency of 31.88-45.2 msecs since it involves updating the redo log at a quorum of geo-distributed sites.

Failover. In Metric, failover involves a different Metric process acquiring ownership (AO) over the tables held by the previous owner of the tables through table-level locks and then populating its local DB (PLD) with the latest records of those tables present in the EC redo log (see §3.3). The AO and PLD operations have not been shown in Figure 4(b), since for our experiments and use-cases these operations are only performed during initialization and failover and not during fault-free executions of transactions. However, to provide a preliminary view of the cost of failover, we executed the same transaction used in the latency microbenchmarks for updating 10 rows 100 times, giving one entry in the redo log with 100 elements in the tx-queue. We then performed the AO and PLD operations, which resulted in a mean latency of ∼460 msecs for AO and a mean latency of ∼3 secs for PLD. The results for 50 rows were similar.

7.3 Camunda Benchmarks

The Camunda performance suite runs basic BPMN workflows on the Camunda engine, where each workflow is configured with the number of threads and the number of execution runs. At the end of a test, the suite returns the total latency required to execute all the runs. Using this information, we show the mean throughput (#runs/total latency) and mean latency for different solutions on different sites in Figure 5(a) and Figure 5(b), respectively. While we experimented with different workflows and configurations, here we only present the results for the "SequencePerformanceTest.syncSequence15Steps" (SS15) with 12 threads and 50 runs since the different systems achieved their highest throughput for this workflow. To exercise multiple writes to the DB, this workflow is configured so that Camunda maintains a history of operations in the DB. The SS15 test performs 31 queries: 1 select, and 30 inserts. The results for other workflows are similar.

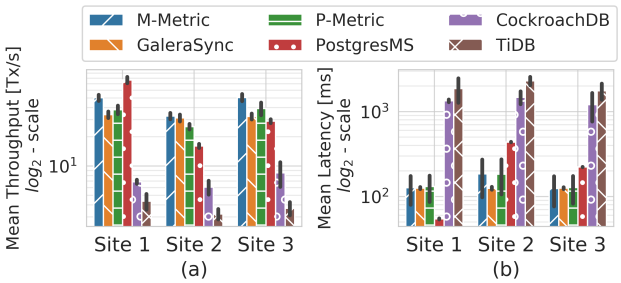


Figure 5: Camunda Benchmarks: Mean throughput and mean latency per site.

When compared with PostgresMS, P-Metric achieves 2.2 and 1.92 times higher mean throughput on Sites 2 and 3, while PostgresMS achieves 1.63 times higher mean throughput on Site 1. Similarly, when compared with GaleraSync, M-Metric achieves 65% and 68% higher mean throughput on Sites 1 and 3, and comparable mean throughput on Site 2. The reasons for these trends are similar to those described in our discussion on latency microbenchmarks.

Both Metric solutions achieve 3.98-7.24x and 7.63-13.99x higher mean throughput than CockroachDB and TiDB, respectively, across all the sites. The inverse of these trends are observed in Figure 5(b). Metric's superior performance over these solutions is mainly because the owners/leaseholders of the key-ranges were spread across sites for all

the runs in both CockroachDB and TiDB. This often adds an additional cross-site access to operations when compared to Metric, as explained in §7.2.

7.4 TPC-C Benchmarks

TPC-C simulates a warehouse-centric order processing system consisting of 9 tables and 5 procedures, where all transactions have an associated warehouse id. We use the default ratios for all the procedures, viz., 45%, 43%, 4%, 4%, and 4% for "NewOrder","Payment", "OrderStatus", "Delivery", and "StockLevel", respectively. Using OLTP-bench we run the TPC-C benchmark and evaluate the throughput and latency as measured at the OLTP process on each site. These transactions have a mixture of both reads and writes. We evaluate the impact of 10 warehouses[9] to increase the data-set and the range of queries, and perform each experiment 5 times. We first present results where OLTP processes access different tables on different sites with no distributed transactions, i.e., there are independent copies of the 9 tables on each site. We then present results where the processes access a common set of 9 tables across sites.

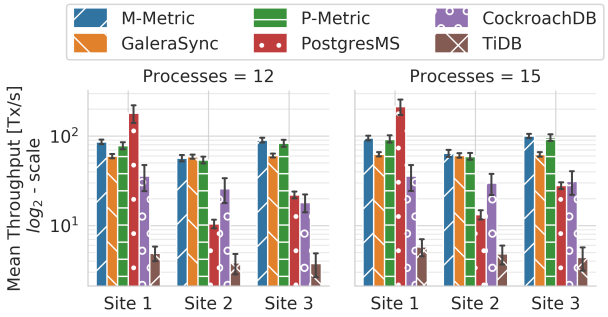


Figure 6: TPC-C: Mean throughput per site where OLTP processes access different tables.

Different tables on different sites. In Figure 6, we present the mean throughput with standard deviation bars for each solution with 12 and 15 OLTP processes. When compared with PostgresMS, P-Metric achieves [4.2x, 5.2x] and [3.5x, 3.8x] higher mean throughput on Sites 2 and 3, while PostgresMS achieves [2.3x, 2.5x] higher mean throughput on Site 1. The last result is mainly because the master is located on Site 1. Similarly, when compared with GaleraSync, M-Metric achieves [1.38x, 1.52x] and [1.44x, 1.61x] higher mean throughput on Sites 1 and 3, while GaleraSync achieves a comparable mean throughput to M-Metric on Site 2. The last result follows mainly because Site 2 is close to both Sites 1 and 3 in terms of latency, and the gains achieved by M-Metric in waiting only for a quorum of responses are largely overridden by the overhead of fetching from a local tx-queue and writing to the redo log). Both Metric solutions achieve 2.11-5.2x and 12.94-26.21x higher mean throughput than CockroachDB and TiDB, respectively, across all the sites. The inverse of these trends are reflected in the site-specific latency CDFs shown in Figure 7 that correspond to the experiments in Figure 6 with 15 OLTP processes. The reasons for these trends are similar to those described in §7.3.

Same tables on different sites. In Figure 8, we present the mean throughput and mean latency per site for 12 OLTP processes; trends for 15 processes were similar and the latency CDF did not yield new insights. When

[9][39] shows similar trends for 6 warehouses.

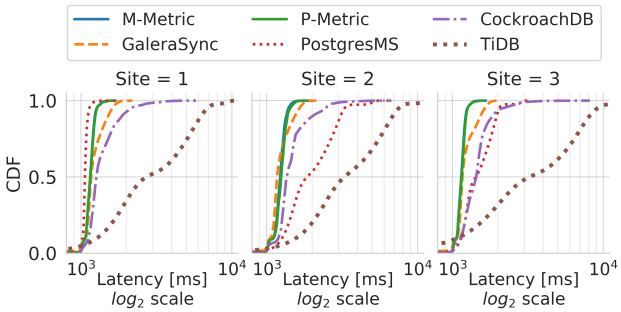


Figure 7: Latency CDF for Figure 6 (P- and M-Metric are indistinguishable.)

the same 9 tables are accessed by OLTP processes across sites, GaleraSync, PostgresMS, CockroachDB, and TiDB achieve 18.58-32.08x, 8.63x-30.79x, 3.29x-4.58x, and 1.72x-2.79x higher mean throughput, respectively, than both the Metric solutions across all the sites. The inverse of these trends are observed in Figure 8(b). This is primarily because, unlike other solutions, every time a table is accessed in Metric across different sites, ownership needs to transition and the local DB on that site needs to be populated with the latest data—a time consuming process, as evidenced by the AO and PLD numbers in §7.2. CockroachDB and TiDB in particular have been optimized for such access patterns with conflict-resolution and fine-grained record-level locking to ensure high performance. While our current use-cases do not require it, we discuss several mitigating strategies in §9 to support future use-cases with such access patterns.

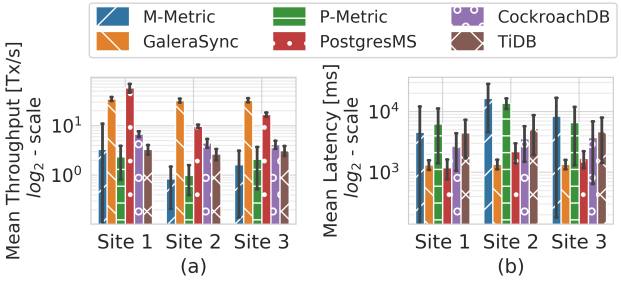


Figure 8: TPC-C: Mean throughput and mean latency per site, where OLTP processes access the same tables.

8. RELATED WORK

Several existing middleware systems support DB clustering through a JDBC interface, but these either do not provide multi-master strict serializibility, require extensive annotation of service code, or both. An example is C-JDBC [15], which provides a single virtual DB on top of a collection of heterogeneous DBs. It strict serializability in single master or active-passive architectures, where each transaction is processed by a single controller and each update is propagated to all back-end DBs that contain the tables. While non-overlapping transactions can be executed on different controllers, they leverage explicit knowledge of service code to use this feature. Metric, which is specifically tailored for geo-distribution, supports a multi-master architecture in which any Metric process can execute any transaction and only commits of a transaction needs to be communicated to a quorum of replicas. Another example is Gyro [30], which

provides multi-master strict serializability with high performance across sites using code annotation to identify transactions that can be executed fully on a local DB instance. In contrast, Metric is a drop-in solution that leverages use-cases in which there are very few overlapping transactions across sites to provide the same properties without requiring any change or annotations to the service code. Apache Ignite [34] also serves as a transparent middleware that can be used on top of any JDBC-compliant DB. However, this system only acts as a caching layer and relies on the existing DB's clustering solution.

Certain aspects of enterprise DBs are also related to the type of functionality realized by Metric. In particular, since transactions in our use-cases are partitioned across sites, it is possible to build a DB solution for these by deploying Independent Master-Slave Clusters (IMSC) configured for synchronous replication across the slaves where a service replica sends all requests to the closest master replica [49, 37, 46]. While IMSC is conceptually simpler, Metric has two major advantages. First, to provide a replication or scale-out factor of r across s sites, IMSC requires $s \cdot r$ DB replicas while Metric requires only s DB replicas and r Cassandra replicas. From an operational point of view, it is far harder to size, maintain, and deploy all these additional stateful replicas in IMSC. Second, while Metric is optimized for access patterns where there is no overlap between DB records across sites, it is still capable of supporting such transactions. For example, while in Metric a service replica can read and write to the same DB record at different sites, IMSC cannot support such access patterns since the masters belong to independent clusters with no shared DB records. Such transactions in Metric currently incur a higher cost, but as discussed below in §9, there is a clear path towards improvement.

Like Metric, other existing DB solutions are specifically optimized for geo-distribution [19, 17, 56, 66, 48, 27, 18, 58]. While other individual differences are outlined below, the most important distinction between all of these systems and Metric is that none are drop-in solutions, i.e., a service has to replace its existing DB. As mentioned in §1 and §5, this is often impractical for production use-cases, especially those that rely on third party software that use a specific DB. However, unlike the table-level locking used by Metric, these solutions use techniques like conflict-resolution and locking at the record level to improve performance for transactions with overlapping records across sites. Even though Metric currently performs well within its targeted domain, as its use expands we will leverage ideas from these systems to further optimize our solution (see §9.)

Our use of EC store locks is similar to the use of locks over data tablets in Spanner [19] or key-value maps in CockroachDB [17]. However, as described in §3.1, Metric uses the EC store's higher-level abstraction of critical sections that handle failures rather than constructing redo logs from first principles, a far more complex and error prone process. Further, the EC store provides semantics that are tailor-made for realizing a drop-in solution. SLOG [56] also exploits locality of data-access patterns to achieve serializability without compromising on performance across sites. However, it assumes that the full transaction is available before processing it and further, require knowledge of what data is accessed by the transaction. While Metric is most efficient when different replicas are processing non-overlapping transactions, it does not require any advance planning. PNUTS [18, 58] provides a geo-distributed store with a per-record timeline consistency model, where a master replica assigned to each record based on locality enforces consistent read/writes for that record. Like Metric, PNUTS uses an externalized log and exploits predictable locality of writes to optimize performance. However, PNUTs does not provide a standard SQL interface nor does it support consistent updates to multiple records across tables. Metric's use of table-level ownership is similar to G-store's [20] key-group ownership abstraction, but the latter only provides multi-key transactional access for single-site usage, as opposed to Metric's abstraction of a serializable replicated DB.

9. FUTURE WORK

Tunable isolation Levels. While in this paper we focus on providing strict serializability, we intend to explore providing a range of isolation levels such as asynchronous replication, read committed, and repeatable reads through selective use of locks in the redo log. Further, we could make this configurable at a per-table level. Thus, for example, if a service only requires asynchronous replication for certain DB tables, then operations on these tables could proceed without requiring locking the redo log.

Reducing cost of ownership. As mentioned above, the current use-cases for Metric typically issue non-overlapping transactions across different Metric processes, and hence ownership transition happens only during initialization and failure. However, we are currently working on reducing the cost of ownership acquisition for use-cases for which the transition is more frequent. Our approach is based on three broad ideas: (1) richer ownership semantics over tables such as separate read and write ownership to provide more concurrency (e.g., multiple concurrent readers), (2) finer-grained ownership over different parts of a table with non-overlapping access patterns (e.g., concurrent ownership of non-overlapping views in a table), and (3) mining usage patterns to prevent premature release of ownership.

Building a standalone DB. While our focus here has been to provide a drop-in solution for DB clustering, the performance advantage for our usage patterns over CockroachDB and TiDB has motivated us to pursue building a standalone DB based on Metric. In this solution a high performance in-memory DB such as SQLite [60] would be used in place of MariaDB or PostgreSQL.

10. CONCLUSIONS

Production geo-distributed services at AT&T and elsewhere often rely on DBs that are not optimized for multi site performance. In this paper, we present a novel, yet pragmatic approach to this problem in the form of a drop-in middleware called Metric that uses an existing service's DB in conjunction with a cross-site clustering solution specifically tailored for geo-distribution. Metric's significant advantage lies in how easily it can be integrated with existing services—not a single line of their original code needs to be changed. We present the novel design of Metric based on an entry-consistent redo log and prove that it achieves strict serializability. For the access patterns common in our use-cases, Metric outperforms the clustering solutions of widely-used DBs, and work is underway on enhancing the performance of Metric for other use-cases through tunable isolation levels and fine-grained ownership semantics. The open-source implementation of Metric is being integrated with multiple AT&T production services with more in the planning stage.

11. REFERENCES

[1] Apache Avatica. https://calcite.apache.org/avatica.

[2] Aws inter-region latency monitoring. https://www.cloudping.co/.

[3] B. Balasubramanian, R. D. Schlichting, and P. Zave. Brief announcement: MUSIC: multi-site entry consistencyfor geo-distributed services. In C. Newport and I. Keidar, editors, *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing, PODC 2018, Egham, United Kingdom, July 23-27, 2018*, pages 281–284. ACM, 2018.

[4] Beautiful business process management with camunda bpm. https://camunda.com/.

[5] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Digest of Papers. Compcon Spring*, pages 528–537, Feb 1993.

[6] Business process model and notation. http://www.bpmn.org/.

[7] pache Calcite. https://calcite.apache.org/.

[8] Camunda data model. https://docs.camunda.org/manual/7.5/user-guide/process-engine/database/.

[9] The Camunda Process Engine Performance Test Suite. github.com/camunda/camunda-bpm-platform/tree/master/qa/performance-tests-engine.

[10] Sorting in Cassandra. https://www.datastax.com/blog/2015/03/we-shall-have-order.

[11] Cassandra's Time-based UUIDs. https://docs.datastax.com/en/archived/cql/3.3/cql/cql_reference/uuid_type_r.html.

[12] CockroachDB Multi-region Topology Patterns. https://www.cockroachlabs.com/docs/stable/topology-patterns.html#multi-region-patterns.

[13] Deploy CockroachDB On-Premises. https://www.cockroachlabs.com/docs/stable/deploy-cockroachdb-on-premises-insecure.html.

[14] Performance Benchmarking CockroachDB with TPC-C . https://www.cockroachlabs.com/docs/stable/performance-benchmarking-with-tpc-c-1k-warehouses.html.

[15] E. Cecchet, J. Marguerite, and W. Zwaenepole. C-jdbc: Flexible database clustering middleware. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 26–26, Berkeley, CA, USA, 2004. USENIX Association.

[16] Cloudify: Pure-play cloud orchestration and automation based on tosca. http://getcloudify.org/.

[17] CockroachDB. www.cockroachlabs.com.

[18] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *PVLDB*, 1:1277–1288, 08 2008.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] S. Das, D. Agrawal, and A. Abbadi. G-store: A scalable data store for transactional multi key access in the cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*, pages 163–174, 7 2010.

[21] DCAE. https://wiki.onap.org/display/DW/Data+Collection+Analytics+and+Events+Project.

[22] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[23] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, Jan. 1987.

[24] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988.

[25] Metric code. https://gerrit.onap.org/r/gitweb?p=music/mdbc.git;a=summary, 2019.

[26] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.

[27] FoundationDB. https://apple.github.io/foundationdb/.

[28] MariaDB Galera Cluster. https://mariadb.com/kb/en/library/what-is-mariadb-galera-cluster/.

[29] About Galera Replication. https://mariadb.com/kb/en/about-galera-replication/.

[30] Gyro: A modular scale-out layer for single-server DBMSs, in proceedings of SRDS 2019 (citation yet to be published.

[31] he Transaction Processing Council. Tpc-c benchmark (revision 5.11.0). http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf.

[32] S. Hemminger. Network emulation with NetEm. In M. Pool, editor, *LCA 2005, Australia's 6th national Linux conference (linux.conf.au)*, Sydney NSW, Australia, Apr. 2005. Linux Australia, Linux Australia.

[33] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[34] Apache Ignite. https://ignite.apache.org/.

[35] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[36] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[37] MariaDB's Semi-synchronous Replication. https://mariadb.com/kb/en/semisynchronous-replication.

[38] MariaDB JDBC Driver. https://mariadb.com/kb/en/about-mariadb-connector-j/.

[39] Metric Technical Report. https://www.dropbox.com/s/1ehqph0r23z3l8p/METRIC_VLDB_Techreport_2020.pdf.

[40] Music onap. https://wiki.onap.org/display/DW/MUSIC-Multi-site+State+Coordination+Service, 2019.

[41] Nagios - the industry standard in it infrastructure monitoring. https://www.nagios.org/.

[42] Onap service orchestrator. https://wiki.onap.org/pages/viewpage.action?pageId=1015834.

[43] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, pages 305–319, June 2014.

[44] Open network automation platform (onap). `https://www.onap.org/`.

[45] Openjdk jmh. `https://openjdk.java.net/projects/code-tools/jmh/`.

[46] Oracle Replication Overview. `https://docs.oracle.com/cd/A87860_01/doc/server.817/a76959/repover.htm`.

[47] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[48] Pingcap TiDB. `https://www.pingcap.com/en/`.

[49] PostgreSQL Synchronous Replication. `https://www.postgresql.org/docs/9.5/warm-standby.html#SYNCHRONOUS-REPLICATION`.

[50] PostgreSQL. `https://www.postgresql.org/`.

[51] R. Potharaju and N. Jain. An empirical analysis of intra- and inter-datacenter network failures for geo-distributed services. In *ACM SIGMETRICS Performance Evaluation Review*, volume 41, pages 335–336, 06 2013.

[52] L. F. C. Project. Opendaylight: Open platform for network programmability to enable sdn and create a solid foundation for nfv. `http://www.opendaylight.org/publications/opendaylight-open-source-community-and-meritocracy-software-defined-networking`.

[53] Protocol Buffers (protobuf). `https://developers.google.com/protocol-buffers`.

[54] F. Razzoli. *Mastering MariaDB*. Packt Publishing, 2014.

[55] B. Reed and F. P. Junqueira. A simple totally ordered broadcast protocol. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 2:1–2:6, New York, NY, USA, 2008. ACM.

[56] K. Ren, D. Li, and D. J. Abadi. Slog: Serializable, low-latency, geo-replicated transactions. *PVLDB*, 12(11):1747–1761, July 2019.

[57] E. Saurez, B. Balasubramanian, R. Schlichting, B. Tschaen, S. Puzhavakath, and U. Ramachandran. Metric: A middleware for entry transactional database clustering at the edge. In *Proceedings of the 3rd Workshop on Middleware for Edge Clouds and Cloudlets*, MECC'18, page 2–7, New York, NY, USA, 2018. Association for Computing Machinery.

[58] A. Silberstein, J. Chen, D. Lomax, B. McMillan, M. Mortazavi, P. P. S. Narayan, R. Ramakrishnan, and R. Sears. Pnuts in flight: Web-scale data serving at yahoo. *IEEE Internet Computing*, 16(1):13–23, 2012.

[59] Software defined network controller project. `https://wiki.onap.org/display/DW/Software+Defined+Network+Controller+Project`, 2019.

[60] SQLite. `https://www.sqlite.org/`.

[61] SQL UUIDs. `https://oracle-base.com/articles/9i/uuid-9i`.

[62] The java database connectivity. `https://www.oracle.com/technetwork/java/javase/jdbc/index.html`, 2019.

[63] The network is reliable: An informal survey of real-world communications failures. http://www.bailis.org/papers/partitions-queue2014.pdf.

[64] Deploy TiDB Using TiDB Ansible. `https://pingcap.com/docs/stable/online-deployment-using-ansible/`.

[65] How to Run TPC-C Test on TiDB . `https://pingcap.com/docs/stable/benchmark/benchmark-tidb-using-tpcc/`.

[66] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvilli, and X. Bao. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 789–796, New York, NY, USA, 2018. ACM.

[67] Zlib Compression. `https://www.zlib.net/`.