GRIT: Consistent Distributed Transactions across Polyglot Microservices with Multiple Databases

Guogen Zhang
eBay Inc.
San Jose, USA
genzhang@ebay.com

Kun Ren
eBay Inc.
San Jose, USA
kuren@ebay.com

Jung-Sang Ahn
eBay Inc.
San Jose, USA
junahn@ebay.com

Sami Ben-Romdhane
eBay Inc.
San Jose, USA
sbenromdhane@ebay.com

***Abstract*—The popular microservice architecture for applications brings new challenges for consistent distributed transactions across multiple microservices. These microservices may be implemented in different languages, and access multiple underlying databases. Consistent distributed transactions are a real requirement but are very hard to achieve with existing technologies in these environments. In this demo we present GRIT: a system that resolves this challenge by cleverly leveraging deterministic database technologies and optimistic concurrency control protocol(OCC). A transaction is optimistically executed with its read-set and write-set captured during the execution phase. Then at the commit time, conflict checking is performed and a global commit decision is made. A logically committed transaction is persisted into logs first, and then asynchronously applied to the physical databases deterministically. GRIT is able to achieve consistent, high throughput and serializable distributed transactions for any applications invoking microservices. The demonstration offers a walk-through of how GRIT can easily support distributed transactions across multiple microservices and databases.**

***Keywords*-microservice, distributed transactions, OCC, deterministic**

I. Introduction

Microservice architecture [1] is widely used in large-scale cloud data platforms and application development. Microservice architecture provides flexibility for application development and reuse of fine-grained services. Microservices can be developed by different domain teams to support business applications. They may be implemented in various languages, such as Java or Golang, and access multiple underlying databases. Applications in a microservice architecture usually require invocation of multiple microservices, which access multiple databases. When an application invokes multiple microservices, it needs distributed transactions to make consistent updates to underlying databases. However, how to support consistent distributed transactions in scale-out databases is a well-known challenge, and is even more challenging in a microservice architecture.

It is possible that we implement some scalable distributed transaction support with a special language, and dictate that applications be written in this special language (such as SQL/PSM [12]). But this will not be compatible with the microservice architecture. In addition, there could be complex logic in microservice and application implementation. Developing such a language is not a small effort. Furthermore, applications would have to be rewritten in such a new language, defeating the purpose of microservice architecture.

The traditional technique is to use two-phase commit (2PC) protocol [10] to achieve distributed transactions. Unfortunately it does not work well in large-scale high-throughput systems, in particular for the applications that have a lot of transaction conflicts [2]. The reason is that locks are held during the entire 2PC process that significantly increase the transaction conflicts and latency. Other methods include persistent message queue pattern for loosely coupled distributed transactions [13], which requires some framework and application logic to compensate failed transaction steps or even business policy to remediate through business measures, costing business money and impacting user experiences. Systems like Spanner [4], YugaByte [8], FoundationDB [7] and CockroachDB [9] can achieve distributed transactions on a single database, which cannot be applied to cross multiple microservices.

Recently deterministic database systems Calvin [2] [3] and FAUNADB [5] were proposed, they are able to scale distributed transactions without 2PC protocol. The idea is that all transactions are ordered using a global Paxos-based [6] log before execution, then all replicas will follow this global order to deterministically execute the transactions using deterministic concurrency control protocol. If a transaction's read/write sets are known, then the transaction is simply put into the global transaction log. Otherwise, it needs to send read-only reconnaissance query that performs all the necessary reads to discover the transaction's full read/write set before being put into the global transaction log. During the actual transaction execution, the transaction might be aborted and restarted if the "reconnoitered" read/write set is no longer valid. This is possible because the records read in the reconnaissance phase might be changed by other transactions.

This approach is able to achieve higher transaction throughput because of simplified replication protocol and deadlock avoidance. Unfortunately this approach still needs one-phase commit for distributed transactions to gather commit/abort decisions from all involved data shards and make the final transaction commit/abort decision, and the locks are held during the one-phase commit that increases the conflict contention. In addition, these systems only know the commit/abort decision during the actual execution phase which significantly increase latency for aborted transactions.

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00230

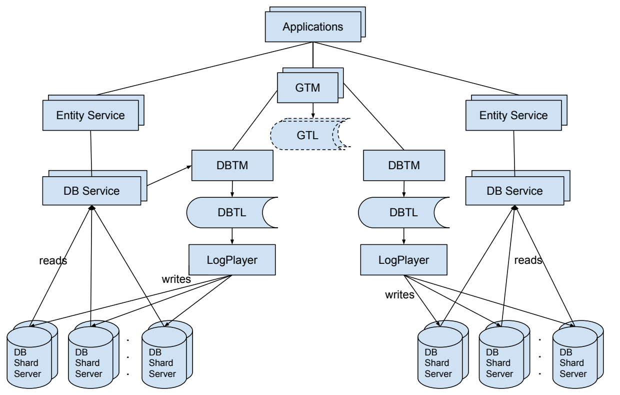


Fig. 1. System Architecture Illustated with Two Databases

GRIT leverages some deterministic ideas, such as ordering transactions in Paxos-based logs before execution. Similar to reconnaissance phase in Calvin, we perform the OCC execution phase [11] which optimistically executes the transaction logic in microservices and saves the temporary write results. Our system is distinct from Calvin/FAUNADB in that we use our novel deterministic conflict resolution algorithm to resolve conflicts immediately after the OCC execution phase, then we can know the local abort/commit decision, and perform quick global commit coordination before persisting it into the commit logs. Obviously GRIT can significantly reduce the transaction latency for aborted transactions. This also indicates that only committed transactions go to the transaction logs, and no further execution is needed during the transaction log execution (physical materialization). Furthermore, the log play phase (corresponding to Calvin's execution) becomes pretty simple, just to apply the write operations that are already calculated during the OCC execution phase. That's why we call it physical materialization of transactions.

In this demo, we showcase GRIT: a novel system that supports consistent distributed transactions across microservices that are implemented in various languages and use multiple underlying databases. The system is efficient and scalable, and provides serializability for transactions across multiple databases exposed through microservices. We use a simplified application scenario to illustrate the challenges and solutions we propose.

II. Architecture and Design

We demonstrate GRIT, a system that supports distributed transactions across microservices with multiple underlying databases. The architecture of GRIT with two databases is illustrated in Figure 1. The databases can be partitioned in a scale-out deployment, which is not our focus in this demo. The key components of GRIT include:

- **GTM**: Global Transaction Manager. It coordinates global transactions across multiple databases covered by multiple DBTMs. There can be one or more GTMs.
- **GTL**: Global Transaction Log. It represents the transaction request queue for a GTM. The order in a GTL determines the relative serializability order among global transactions. Persistence of GTLs is optional.
- **DBTM**: Transaction Manager at each database realm. The conflict checking and resolution, i.e. local commit decision, is located here. A database realm is the scope of a database covered by this DBTM for conflict checking. It can be a database, a shard or multiple shards of the database (we use database to mean database realm for simplicity). There can only be one DBTM for a database.
- **DBTL**: Transaction Commit Log for a DBTM at each database. It logs logically committed transactions that relate to this database (including single database transactions and multi-database transactions). The transaction order in a DBTL determines the serializability order in the database system, and global transactions from GTLs are reflected here. It is analogous to a write-ahead log (WAL) for committed transactions in a traditional database.
- **Log Player**: It pushes commit log entries to their target database shard servers for write execution. Its role is the same as that of a log replication.
- **DB Shard Servers**: Deterministic database engines. Each server uses deterministic concurrency control to materialize logically committed transactions concurrently. It also need to support multi-versioning and snapshot reads for isolation.

There are some other components in our system:

- **Microservice**: building blocks to provide business-oriented service for applications to implement business logic. Each DB may have support for multiple microservices, and executions of microservices are independent of each other.
- **DB Service**: Provide DB server read/write interface and directly access DB Servers. It also caches the read/write sets of each transaction during the execution phase and sends them to its DBTM for conflict resolution at the commit time. There is no limit on how many DB service instances a system can have.

We leverage the well-known optimistic concurrency control (OCC) for execution of microservices and application logic. And at the commit time, the system performs conflict resolution in DBTMs and makes global commit decision in GTMs, and orders committed transactions in DBTLs to be materialized by the underlying deterministic databases.

The consistency of distributed transactions are guaranteed through three phases: **optimistic execution phase**, **logical commit phase**, and **physical materialization phase**, as illustrated in Figure 2. Efficiency is achieved by separating the logical commit decision from physical materialization. A transaction is considered committed once its effect is persisted into the transaction commit logs. And physical materialization to the databases is out of the commit decision loop.

Now we describe the details of the three phases:

1) **The optimistic execution phase**: a transaction fetches

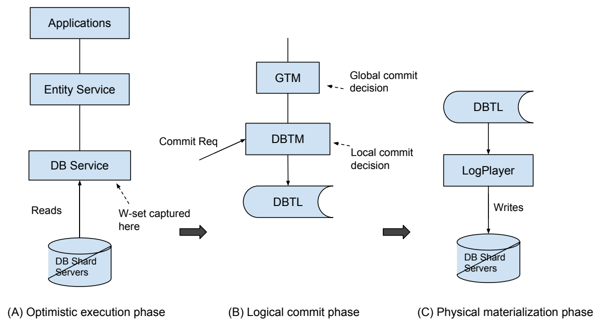


Fig. 2. Major components involved in each phase of a distributed transaction

and updates databases through microservices and database services, the fetches and updates are captured by each database service as (r-set, w-set). Note that we also need to capture the version info (i.e. Log Sequence Number - LSN) for each data item in the r-set for conflict resolution.

2) **The logical commit phase**: At the commit time, conflict resolution is performed by transaction managers (both DBTMs and the GTM). If there is a conflict, the transaction is aborted. Otherwise, it's committed logically and persisted into transaction logs (DBTLs).
3) **The physical materialization phase**: Databases will materialize the transactions from the commit logs (DBTLs) and make physical commit to the databases.

The logical commit decision is accomplished at two levels as described below.

- **At a DB level**: on receiving a commit request, the DB service agent submits the request with its local (r-set, w-set) and meta information for the transaction to the responsible DBTM. The DBTM performs conflict checking based on its w-set cache of recently committed transactions. The logic of checking conflict is similar to that in the traditional OCC except that the DBTM does not access DB to figure out the conflicts but uses a cache of recent updates. If there is no conflict, it can be locally committed. And if the transaction only involves a single database, then the transaction's w-set will be appended into the DBTL log with an assigned LSN, and the transaction is logically committed. If the transaction involves multiple databases, the DBTM sends its local commit decision to the GTM for the transaction, and acts based on the response from the GTM. If there is a conflict during conflict checking, or the global commit decision from the GTM is "ABORT", the transaction is aborted. The application needs to retry.
- **At global level**: Commit of a transaction involving multiple databases has to be coordinated by a GTM. On receiving the commit request for a transaction from the application, the GTM will wait for the involved DBTMs' commit decisions. If all the commit decisions from the DBTMs are "COMMIT", the transaction can be committed. If any DBTM reports "ABORT", the transaction is aborted. The GTM informs the DBTMs of the global commit decision by responding to their submissions. This interaction is simpler than a 2PC protocol and requires no locking in the databases.

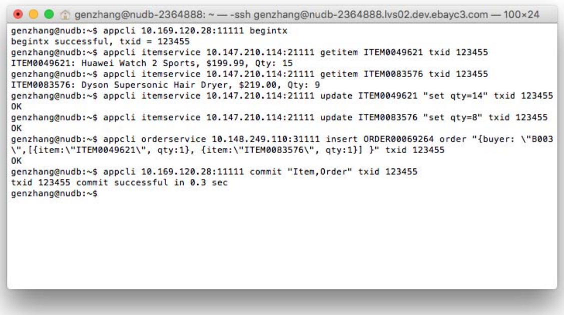
For physical materialization phase, we leverage the deterministic database engines to achieve that. Under normal conditions, the Log Player will steam log entries sequentially to target database shard servers, and transaction writes are deterministically executed following the transaction order in the DB level transactions logs (DBTLs). Under abnormal conditions when the updates cannot be performed(hardware errors, software crash etc), recovery has to be performed on the particular shard server and we can leverage the deterministic recovery algorithm which is much simpler than traditional algorithms.

It is worth noting that conflict checking at a DBTM is sufficient with the cache of w-sets from all the recently committed transactions, as long as all the updates go through the same DBTM for the covered scope of the database. The goal of conflict checking is to see if there is any other transaction that has changed an entry since the transaction read it. A transaction reads from DB servers directly in the OCC execution phase, which include all the commits from transactions up to the last log entry that has been materialized at the time of reads. Each database maintains the Last_Commit_LSN for itself. The Last_Commit_LSN is remembered for a transaction when it starts read, called transaction's Read_LSN on this database. A transaction only needs to check those w-set entries cached after its Read_LSN for its r-set entries. If there is none, that means all reads are fresh and the transaction can be committed. A cached w-set entry is only useful if there is a transaction that would potentially conflict with it, thus can be purged if its LSN is less than the oldest Read_LSN (Oldest_Read_LSN) of inflight transactions on the database.

Various read isolation levels can be provided. Each log LSN defines a logical snapshot for the database. Local snapshot reads can be supported by given a specific LSN for each database. Strongly consistent read at a global consistent snapshot can be provided by registering a read-only transaction through a GTM, and the GTM can request LSNs from involved DBTMs for a consistent snapshot across multiple databases. We do not expand this flow here. For a read-modify-write transaction, it always reads the latest committed data.

III. Demonstration Scenarios

We demonstrate the system by a simple simulated application involving purchasing items on an ecommerce website on the server side. The application takes an order that contains a list of (item, quantity) to purchase, and calls two microservices: one is ItemService, which will maintain the listing items and their inventories, the other is OrderService, which will create orders. Microservices are illustrated by two gRPC services: ItemService and OrderService. To illustrate polyglot microservices, we use two different languages to implement



genzhang@nudb-2364888: ~ — -ssh genzhang@nudb-2364888.lvs02.dev.ebayc3.com — 100×24

```
genzhang@nudb:~$ appcli 10.169.120.28:11111 begintx
begintx successful, txid = 123455
genzhang@nudb:~$ appcli itemservice 10.147.210.114:21111 getitem ITEM0049621 txid 123455
ITEM0049621: Huawei Watch 2 Sports, $199.99, Qty: 15
genzhang@nudb:~$ appcli itemservice 10.147.210.114:21111 getitem ITEM0083576 txid 123455
ITEM0083576: Dyson Supersonic Hair Dryer, $219.00, Qty: 9
genzhang@nudb:~$ appcli itemservice 10.147.210.114:21111 update ITEM0049621 "set qty=14" txid 123455
OK
genzhang@nudb:~$ appcli itemservice 10.147.210.114:21111 update ITEM0083576 "set qty=8" txid 123455
OK
genzhang@nudb:~$ appcli orderservice 10.148.249.110:31111 insert ORDER00069264 order "{buyer: \"B003
\",[{item:\"ITEM0049621\", qty:1}, {item:\"ITEM0083576\", qty:1}] }" txid 123455
OK
genzhang@nudb:~$ appcli 10.169.120.28:11111 commit "Item,Order" txid 123455
txid 123455 commit successful in 0.3 sec
genzhang@nudb:~$
```

Fig. 3. Screenshot of a demo window

the two services. The ItemService is implemented with Java, and talks to the service from the database for Items. The OrderService is implemented with Golang, and talks to the service from the database for Orders. The database services are gRPC services implemented using C++. They implement part of OCC logic to capture the (r-set, w-set) for each database for a distributed transaction. They also take transaction log entries and deliver them to the underlying DBMS to materialize.

The application is simulated by a CLI in a terminal that calls microservices within a transaction as many times as needed for items, and then creates an order before commit. A screenshot of the application window is shown in Figure 3. Demo users are able to edit the sample CLI scripts to customize the simulated application data and issue transactions from a terminal.

Another terminal will display information for the GTM current transaction requests for commit. And two other terminals will display information for DBTMs, i.e. transactions and their (r-set, w-set)s from database services and commit status for each database.

To increase concurrency, we will also demonstrate randomly generated transactions from the simulated application submitted from multiple terminals in parallel. We also plan to show the transaction latency and throughput based on different loads.

IV. Novelty and Contributions

It is very hard to support distributed transactions efficiently and that is why many NoSQL database systems don't support distributed transactions, such as Amazon's Dynamo [14], CouchDB [15], Cassandra [16], Bigtable [17] and Azure [18]. However, reducing transactional support results in increased code complexity and development efforts for applications.

We have proposed a novel system to support distributed transactions across microservices involving multiple underlying databases in a scalable setting. It abstracts a transaction with (r-set, w-set) as in OCC transactions and combines many ideas from existing techniques but avoid their shortcomings. Overall it employs logical transaction commit logs and leverages deterministic underlying database engines for performance and scalability. For coordination across databases, we use a mechanism similar to 2PC but apply at logical commit phase, which avoids longer duration. After a transaction is committed logically, they are materialized by the deterministic databases, which are typically scale-out deployment. The logical commit logs are replicated and can replace physical log-based replication.

The key for the scalability and performance is the techniques to avoid coordination during execution phase as well as transaction materialization (physical commit) that are of relatively longer duration. The coordination is at the commit time for conflict resolution that is of short-duration and fast. 2PC-like protocol is only used for cross database transactions, and unlike traditional 2PC where the involved databases typically need to lock the relevant data records during the protocol playout, GRIT doesn't require expensive locking during the protocol. In addition, the coordination is only needed when a transaction truly impacts multiple databases, and single-database transactions only need to go to its local database transaction manager. Deterministic database servers simplify concurrency control and speed up commit materialization process. GRIT is a perfect use of deterministic database engines, which require known (r-set, w-set) to perform deterministic transaction scheduling.

Furthermore, GRIT is able to avoid the implementation of a procedure language (something like PL/SQL, or SQL/PL), which would be a huge undertaking, and the microservice logic itself doesn't need to be changed.

References

[1] https://en.wikipedia.org/wiki/Microservices

[2] Thomson, Alexander and Diamond, Thaddeus and Shao, Philip and Ren, Kun and Weng, Shu-Chun and Abadi, Daniel J, "Calvin: Fast distributed transactions for partitioned database systems", SIGMOD 2012.

[3] Kun Ren, Alexander Thomson, Daniel Abadi, "An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems", VLDB 2014.

[4] J. C. Corbett et al, "Spanner: Google's Globally-Distributed Database", In Proc. of OSDI 2012.

[5] https://fauna.com/.

[6] L. Lamport. Paxos made simple. In SIGACT News, Vol. 32, No. 4, pp. 51-58, 2001.

[7] https://www.foundationdb.org/ .

[8] https://www.yugabyte.com/.

[9] https://www.cockroachlabs.com/.

[10] C. Mohan, Bruce Lindsay and R. Obermarck, "Transaction management in the R* distributed database management system" ACM Transactions on Database Systems (TODS), Volume 11 Issue 4, Dec. 1986, Pages 378 - 396.

[11] H.T.Kung, J.T.Robinson, "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems 6:2, June, 1981.

[12] https://www.iso.org/standard/29864.html.

[13] Pat Helland, "Life beyond Distributed Transactions: an Apostate's Opinion", CIDR 2007.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati et al, "Dynamo: Amazon's highly available key-value store", SIGOPS, 2007

[15] J. C. Anderson, J. Lehnardt, and N. Slater, "CouchDB: The Definitive Guide", 2010.

[16] A. Lakshman and P. Malik, "Cassandra: structured storage system on a p2p network", In PODC, 2009.

[17] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh et al, "Bigtable: a distributed storage system for structured data", In OSDI, 2006.

[18] D. Campbell, G. Kakivaya, and N. Ellis, "Extreme scale with full sql language support in microsoft sql azure.", In SIGMOD 2010.