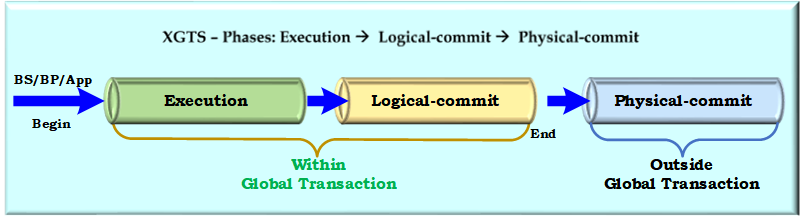
**Executive Summary**

Achieving distributed transactions across microservices is a well-known challenge because microservices run on different machine boundaries and access multiple underlying datastores. This document presents Xelerate Global Transaction Service (XGTS), a new, strategic initiative that solves this problem efficiently and at scale by introducing a global transaction boundary to orchestrate transactions across X3 microservices.

**Primary Objectives:**

* *Handle high transaction throughput* by separating logical-commits (write-ahead log) and physical-commits (datastore commit) by eliminating pessimistic locks and lowering transaction latency
* *Check and resolve transaction conflicts* by making quick, pre-emptive commit/abort decisions
* *Maintain global transaction order* for transaction serializability
* *Execute global transactions* at BP (workflow), BS (step level), and app levels
* *Eliminate expensive, compensating transactions* by maintaining local recovery logs
* *Leverage the X3 platform framework* for atomic transactions using BEs
* *Develop higher-level components* (XGTM, GTL, ETM, ETL, and ERL) around the X3 framework to induce distributed transactional intelligence

This strategic initiative proposes that the timeline of a global transaction be divided into three phases as shown below: execution (1), logical commit (2), and physical commit (3).



Firstly, at the execution phase, each transaction is optimistically executed with all business and data validation and its read-sets and write-sets are captured.

Secondly, at the logical commit phase, conflict-checking and resolution is performed on these read-sets and write-sets and the final commit decision is made by writing only the write-sets of committed transactions into commit-log (write-ahead log) in transactional order.

Lastly, at the physical commit phase, these committed write-sets are physically hardened to their respective datastores, thus maintaining the global transaction order. This phase can be performed independently and asynchronously of the global transaction.

**Introduction**

Microservices are loosely coupled, independently deployable application components that incorporate their own stack, including their own databases and data models and communicate over a network. Supporting distributed transaction across microservices is a well-known challenge because of the heterogenous nature of data being distributed across multiple data stores.

The microservice architecture provides flexibility for the reuse of fine-grained services and is widely used in the development of large-scale applications. This paradigm allows for development by different teams across different domains that each support their own business applications and access their own independent datastores. When an application invokes multiple microservices, it needs distributed transactions to make consistent updates to underlying datastores. However, supporting consistent distributed transactions in scale-out databases is a well-known challenge, and is even more challenging in a microservice architecture setting because each microservice can potentially run on a different machine boundary, use a different data store, and was probably developed by a different domain team using a unique language to support a unique business application.

Most microservices are developed without fully considering the functional needs, scalable demands, and availability of distributed transactions. RDBMS support ACID transactions on data which reside in a single instance but do not support availability on multiple instances. NoSQL/Key-Value datastores don’t support ACID transactions on single instances either.

This clearly puts the onus on application developers to code complex logic and manage these transactions in the application layer. Business entities in the X3 platform already provide guaranteed ACID-compatible, atomic transactions to manage their persistence through the XDM layer onto polyglot datastores. This initiative leverages this capability and builds additional functionality to execute distributed transactions without compromising on data availability, consistency, and fault-tolerance.

**Distributed Transaction Patterns**

* *SAGA Pattern:* This well-known patten uses a persistent message queue for loosely coupled distributed transactions. However, it requires application logic to compensate for failed transactions which cost businesses money and negatively impact users’ experiences.
* *Traditional Two-Phase (2PC) Commit Pattern:* This pattern does not scale well because locks are held during the entire 2PC process and that significantly increases the potential for transactions to conflict, not to mention the increased latency.
* *GRIT Protocol:* This leverages deterministic database technologies and optimistic concurrency control protocol (OCC) by stepping through the same three phases further outlined in this document. This protocol also claims to support high throughput by achieving consistent, serializable distributed transactions for any applications that invoke microservices.

This initiative leverages the GRIT protocol’s concepts and builds distributed transactional intelligence around the X3 platform. Business entities in the X3 platform already provide guaranteed ACID-compatible, atomic transactions to manage their persistence through the XDM layer onto polyglot datastores. The section below describes types of transactions, concurrency control procedures, and provides a detailed overview of the Xelerate Distributed Transaction Manager followed by short description about potential next steps to take.

**Classification of Transactions**

Transactions are typically classified based on their duration, structure, and read and write actions.

**Duration** - Based on duration, transactions are classified as short-life and long-life transactions. Short-life transactions mainly include real-time and online transactions with short execution/response times and affect a relatively small portion of datastore. These cover a large majority of current transaction applications like banking and airline reservation transactions. In contrast, long-life transactions include batch transactions which take a longer time to execute and have access a larger portion of the datastore like statistical applications including but not limited to report generation and image processing.

**Structure –** Based on structure, transactions are classified as flat and nested transactions. Flat transactions consist of a series of instructions embraced between “begin” and “end” markers. Nested transactions have transactions within the main transaction and the operations of a transaction may themselves be other transactions.

**Read and Write Actions –** Based on read and write actions, transactions are classified as two-step, read-before-write, and restricted two-step transactions. Two-step transactions are restricted so that all the read actions are performed before any write actions. Read-before-write enforced data is read before it gets committed and restricted two-step includes both the above.

**Understanding Compensating, Abort, and Recompute Transactions**

**Compensating Transaction:** These are transactions that are initiated by applications to apply corrections to previous work done by earlier transactions. For example, this could be crediting or debiting users for wrong charges. Since these are initiated by applications as separate transactions they are treated just like any other transaction.

**Abort Transaction:** These are also called rollback transactions which need to roll-back within the global transaction boundary. A global transaction involving three microservices executes atomic functionality but the first two microservices have already committed the data and third one failed. This triggers XGTS to issue an abort transaction for first and second microservices.

**Recompute Transaction:** When the system encounters a logical error, all transactions need to be re-computed from some point. This could be from some last transaction or from some previous date. These need to be handled manually on a case-by-case basis.

**Pessimistic & Optimistic Concurrency Control**

Concurrency conflicts occur when the same record in the database is updated by more than one user or application. This has a significant effect on the data and a severe effect down the road. Reversal of these transactions is a costly operation and includes complicated logic. There are 2 solutions to this problem:

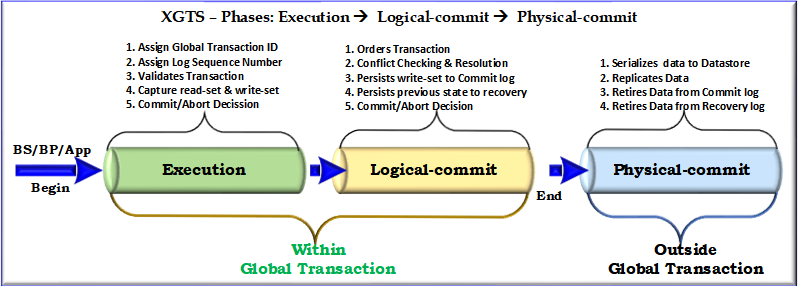
* Avoid this conflict upfront (pessimistic control)
* Fix this conflict later (optimistic control)

**Pessimistic Concurrency Control** (Pessimistic Locking): The system assumes the worst; that two or more users will want to update the same record at the same time, and then prevents that possibility by locking the record, no matter how unlikely conflicts are. The locks are placed as soon as any piece of the row is accessed, making it impossible for two or more users to update the row at the same time. Programming an application with pessimistic concurrency approach can be more complicated and complex to manage because of the risk of deadlocks.

**Optimistic Concurrency Control** (Optimistic Locking): The system assumes that although conflicts are possible, they will be very rare. Instead of locking every record every time that it is used, the system merely looks for indications that two users tried to update the same record at the same time. If that evidence is found, then one user’s updates are discarded, and the user is informed.

**Xelerate Global Transaction Service (XGTS): Overview**

The X3 platform provides the Xelerate Global Transaction Service (XGTS) which is responsible for orchestrating distributed transactions across X3 microservices. These microservices can be running on a single machine or on multiple machine boundaries. XGTS carries out global transactions as shown below in three phases: execution (1), logical commit (2), and physical commit (3).



**Execution** – In this phase, each transaction is optimistically executed with all business and data validation performed by BEs and its read-sets and write-sets are captured. Any invalid data or error is immediately rejected, and a transaction is aborted without any overheads.

**Logical Commit** – In this phase, conflict-checking and resolution is performed on these already captured read-sets and write-sets and the final commit decision is made by writing only the write-sets of committed transactions into the commit log. Since all transaction conflict-checking and resolution-checking happens before logical-commit, only committed transactions are persisted into the commit-log and will significantly reduce the transaction latency for aborted transactions.

**Physical Commit** – In this phase, BEs apply the already validated and conflict-resolved write-sets from committed logs to their respective polyglot datastores using XDM whilst maintaining the transaction order.

By deferring physical commit out of the transaction loop, pessimistic locking during both execution and logical-commit phase will be avoided and will eliminate any waiting for physical commit making this process very efficient.

Unlike SAGA which burdens applications by requiring them to post compensating requests to rollback, this protocol provides a framework for BEs to revert to their previous state from recovery logs with GID within the global transaction boundary.

This approach leverages the X3 platform’s BEs/XDM for business and data validation and captures read-sets and write-sets whilst preserving data integrity and polyglot persistence transparency.

Furthermore, it incrementally builds higher-level abstractions like XGTM, GTL, ETM, ETL, and ERL to induce distributed transactional inteligence. In this context, it identifies a few functional gaps in the X3 platform like logical commit, physical commit, local rollback, local refresh, deadlock abort, timeout abort, and single-be-instance-service that must be implemented in the X3 platform to fully participate in global transactions.

The section below discusses XGTS and its components in detail.

**Key Components of XGTS**

**XGTM**: The Xelerate-Global Transaction Manager orchestrates all global transactions across multiple microservices. Its main responsibilities include the following:

* *Managing the global transaction ID (GID)* namespace and global log sequence number (GSLN)
* *Orchestrating communications* with the multiple microservices
* *Maintaining global persistent states* of each distributed transaction
* *Making commit/abort decisions* at various levels
* *Recovering from various failures* of individual environments

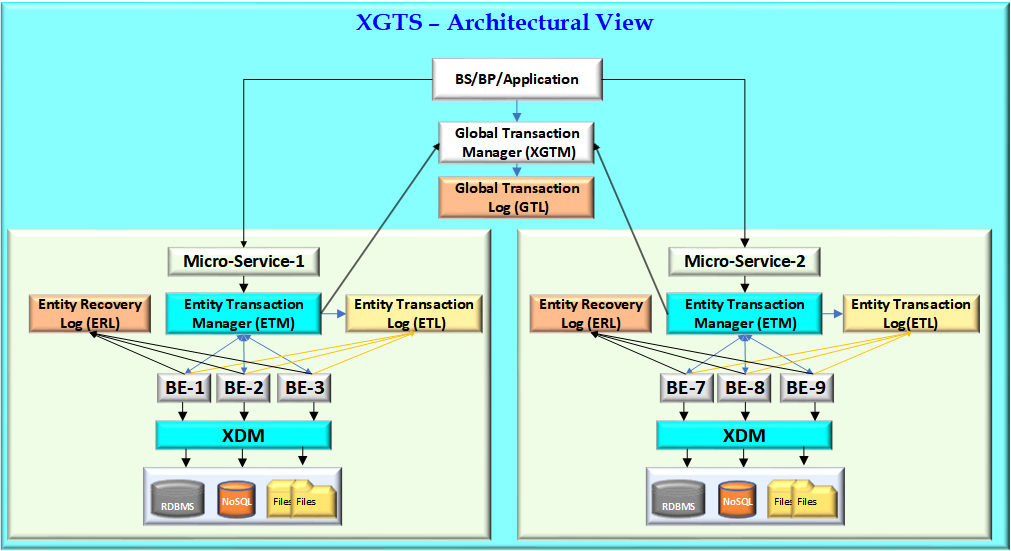
**XGTL**: The Xelerate Global Transaction Log represents the global transaction request queue for a XGTM. The order in a XGTL determines relative serializability order. It assigns a log sequence number (LSN) and all transactions must follow this order.

**ETM**: The Entity Transaction Manager is a local XGTM responsible for orchestrating all local transactions across multiple BEs within a microservice. All local ordering, conflict- checking of transactions, and local commit/abort decisions happens here.

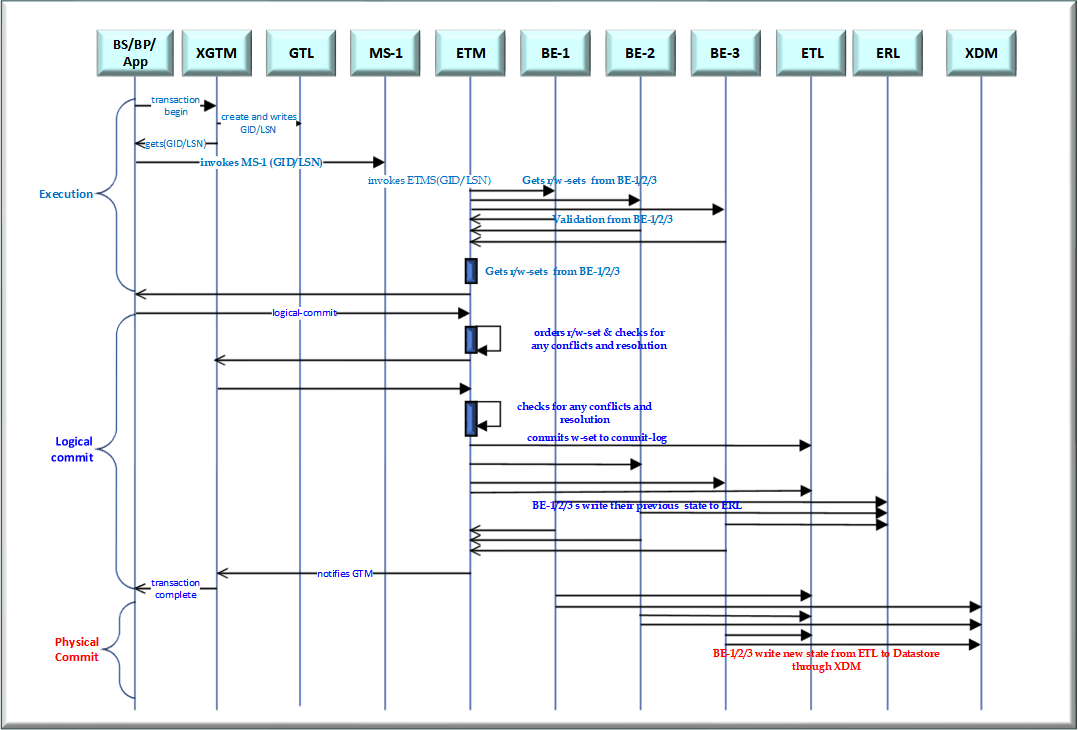
**ETML**: The Entity Transaction Manager Log is analogous to a write-ahead log for ETM which stores only logically committed transactions that relate to this microservice. The transaction order in ETML determines the serializability order for BEs, and global transactions from GTLs are reflected here.

**ERL:** The Entity Recovery Log is the staging area for all microservices and their BEs to write any recovery data to that GID. For example, all BEs participating in GTM write their old/previous states here for any future rollbacks.

**High Level Architecture**



**Global Transaction Execution Order Step-by-Step**



The three-phase execution order of XGTS is described in detail below:

1. **Execution/Prepare Phase**
   1. BP/BS/App initiates a global transaction by requesting a global transaction ID (GID) from XGTM with a list of ETMs.
   2. BP/BS/App invokes microservices with GID.
   3. Microservices invoke ETMs with GID.
   4. ETMs capture read-sets and write-sets from their respective BEs for GID and vote yes for the logical-commit.

**Note**: At this stage if there is any abort decision from BP/BS/App or from BEs because of an exception or timeout, ETMs can simply inform an execution-abort to GTM forget the transaction. No data is committed anywhere.

1. **Logical-Commit Phase** 
   1. BP/BS invokes all ETMs with a logical-commit request for GID.
   2. ETMs order the read-sets and write-sets commit requests, check for any conflicts, and resolve them. Conflict-checking is to see whether there is any other transaction that has changed an entry since the transaction GID has read it.
   3. After checking for any conflicts, ETMs send logical-commit-yes to GTM.

**Note**: At this stage if there is any abort decision from BP/BS/App or from BEs because of an exception or timeout, ETMs can send logical-commit-aborts to GTM to forget the transaction. No data is committed anywhere.

* 1. GTM sends logical-commit-final to ETMs for GID.
  2. All ETMs will check for any final conflicts.
  3. If there are no conflicts, ETMs finally persist write-sets for this GID into ETML (acts as a write-ahead log) with LSN and version.
  4. ETMs also write all the previous states of BEs to ERL along with GID for any rollbacks down the road.
  5. After these two operations, ETMs reports back logical-commit-final-yes to GTM.

**Note**: At this stage if there is any abort decision from BP/BS/App or from ETMs because of final transaction conflicts or timeouts, ETMs report a logical-commit-final-abort decision to GTM, insert the compensating transaction into ETML (if the data is already inserted into ETL and cannot be modified), and remove the log from ETL, thus aborting the transaction.

* 1. GTM collects all logical-commit-final-yes acknowledgements from the ETMs and informs a global commit success/failure decision to BP/BS/App.

1. **Physical-Commit Phase (Outside the Global Transaction Boundary)** 
   1. After the logical-commit, GTM internally initiates a physical-commit to ETMs which in turn invokes the respective BEs to serialise the data from ETML to their respective datastore using XDM. Persistence of the data must ensure guaranteed sequencing based on LSN.
   2. Once the data is finally committed in the respective physical datastore, all traces of the data from ETML and RTL will be removed or archived.

**Touch Points for discussion**

The following three concerns need to be further evaluated:

* Separating logical-commits from physical-commits will give us high transactional throughput. Taking the example of past transactions, it would be a good approach to price transactions in real-time. This would also roll-back transactions locally by reverting them to their old state from the staged log without initiating separate transactions, thus reducing the system overhead.
* Combining logical-commits and physical-commits in the same transaction window might be suitable for batch transactions in which operations are in bulk.
* Physical-commit operations can be performed in bulk and we can use deterministic operations to sync the data to replicas and caches.

**Glossary**

X-GTS Xelerate-Global Transaction Service

X-GTM Xelerate-Global Transaction Manager

GID Global Transaction ID

GTL Global Transaction Log

ETM Entity Transaction Manager

ETL Entity Transaction Log

RTL Recovery Transaction Log

ERL Entity Recovery Log

BE X3 – Business Entity

BS X3 – Business Service

BP X3 – Business Process

BO X3 – Business Operation

XDM X3 – Data Management

**Conclusion**

This document proposes a new, strategic to implement global transaction management in the X3 platform by leveraging concepts from the GRIT protocol and building distributed transactional intelligence, thus extending X3 platform. As business entities in the X3 platform already provide guaranteed ACID-compatible, atomic transactions to manage their persistence through the XDM layer onto polyglot datastores, this proposal incrementally builds higher-level abstractions to induce distributed transactional intelligence at scale.

After we have achieved a teamwide consensus on the approach, each functional component needs to be further detailed with a PoC. This needs to be carefully evaluated, tested, and benchmarked for its throughput, network latency, optimal writes to logs, communication failures, isolation levels, deadlocks, serializability, and data consistency.