# 6 Signs You Might be Misunderstanding ACID Transactions in Distributed Databases

Sid Choudhury | VP, Product

July 22, 2018

As described in A Primer on ACID Transactions, first generation NoSQL databases dropped ACID guarantees with the rationale that such guarantees are needed only by old school enterprises running monolithic, relational applications in a single private datacenter. And the premise was that modern distributed apps should instead focus on linear database scalability along with low latency, mostly-accurate, single-key-only operations on shared-nothing storage (e.g. those provided by the public clouds).

App developers who blindly accept the above reasoning are not serving their organizations well. Enterprises in verticals such as retail, finance, SaaS and gaming can gain a competitive advantage today by launching highly engaging customer-facing apps with transactional integrity, low latency and linear scalability across multiple cloud regions, all simultaneously. And it is possible to do so using both NoSQL/non-relational and SQL/relational APIs.

This post enables app developers to get a more accurate understanding of ACID transactions in distributed databases. It does so by pointing out 6 common signs of misunderstanding that have made their way into developers mindset after 10 years of the NoSQL era.

## 1. ACID Compliance Requires SQL

All SQL variants have dedicated syntax (such as `BEGIN TRANSACTION` & `END TRANSACTION`) to highlight whether a set of operations should be treated as a single ACID transaction. NoSQL APIs usually have no such syntax and updates are usually limited to only one row/key at a time (with eventual consistency). However, increasingly even NoSQL databases are becoming transactional. MongoDB added single shard transactions along with `start_transaction` & `commit_transaction` in its recent 4.0 release. YugaByte DB, a NoSQL compatible database, extended the popular Cassandra Query Language with a SQL-like transaction syntax. Apple's FoundationDB is an ACID-compliant, key-value NoSQL database with a Transaction Manifesto that points out that "transactions are the future of NoSQL."

## 2. ACID Transactions in a Distributed DB Are Always Distributed

ACID transactions in a distributed database are not always distributed. They can be classified into three types, only one of which is distributed.

### Single Row ACID

Search

Top Posts

Announcing YugabyteDB 2.0 (
Jepsen Tested, High-
Performance Distributed SQL

Comparing Distributed SQL
Performance – YugabyteDB vs
Amazon Aurora PostgreSQL vs
CockroachDB

YugabyteDB's Distributed SQL
API Jepsen Test Results

How YugabyteDB Scales to M
than 1 Million Inserts Per Sec

2019 Distributed SQL Summit
Recap and Highlights

Categories

ACID Transactions

Amazon Aurora

Amazon DynamoDB

Amazon Web Services

Apache Cassandra

Apache Kafka

Cloud Providers

CockroachDB

Community News

Company News

Containers

Databases

Distributed SQL

Docker

Ecosystem Integrations

Google Cloud Platform

Google Spanner

Transactions where all the operations impact only a single row (aka key) are called *single row ACID transactions*. A distributed database has to be at the very least strongly consistent in order to support such transactions. A single row is owned by a single shard which is present on a single node. This means for a Replication Factor (RF) 1 deployment, only a single node participates in the transaction. As the RF increases, more nodes come into the transaction and in return increase the fault-tolerance of the deployment.

### Single Shard ACID

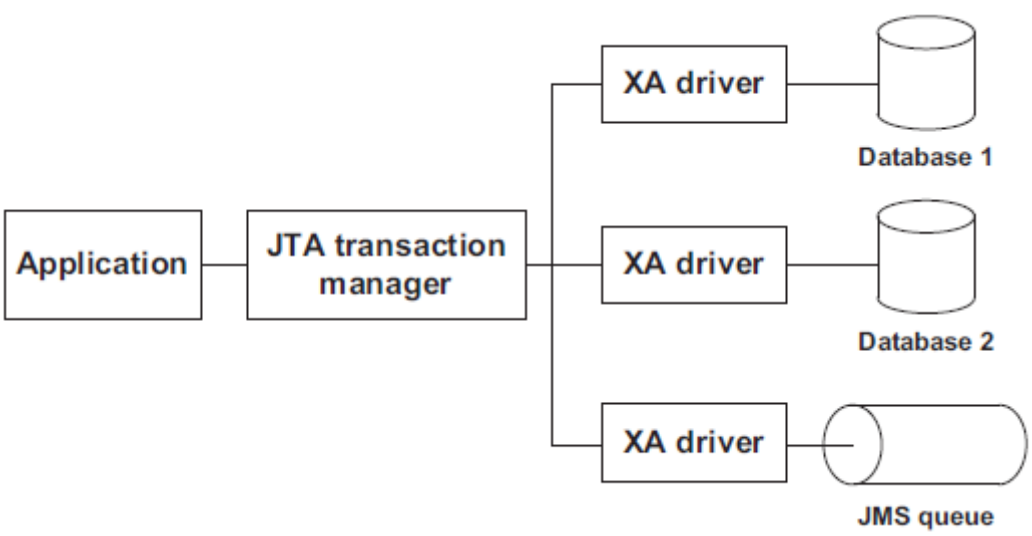
A marginal improvement over single row ACID is *single shard ACID* where all the rows are located in a single shard of a distributed database. Since a single shard is always located inside a single node, this type of transactions are also non-distributed. E.g. MongoDB's latest 4.0 release supports single-shard transactions.

### Distributed ACID

In auto-sharded distributed DBs such as YugaByte DB and Google Cloud Spanner, shards are spread across multiple nodes. Cross-shard transactions in such databases are called *distributed ACID transactions* (or simply *distributed transactions*). Databases with support for this type of transaction also support single row and single shard transactions. However, the opposite is not true. E.g. MongoDB does not support distributed transactions in its multi-shard deployments (called *Sharded Clusters)*.

## 3. Distributed Transactions Today = XA Transactions of the Past

For experienced developers, the term "distributed transaction" usually evokes the painful memories of XA transactions.




How It Works
How To
Jepsen Tests
Kubernetes
Microsoft Azure
Microsoft Azure Cosmos DB
MongoDB
Performance Benchmarks
Pivotal Container Service (P
PostgreSQL
Release Announcements
Spring
Week In Review
YugabyteDB 1.1
YugabyteDB 1.2
YugabyteDB 1.3
YugabyteDB 2.0

*Source: Transaction Management*

XA stands for *eXtended Architecture* and is an X/Open group standard for executing a global transaction in an atomic manner across multiple independent data stores such as databases and persistent queues. A two-phase commit (2PC) protocol is used for the execution. All SQL databases, JMS providers as well as a few transaction managers such as Tuxedo support the standard.

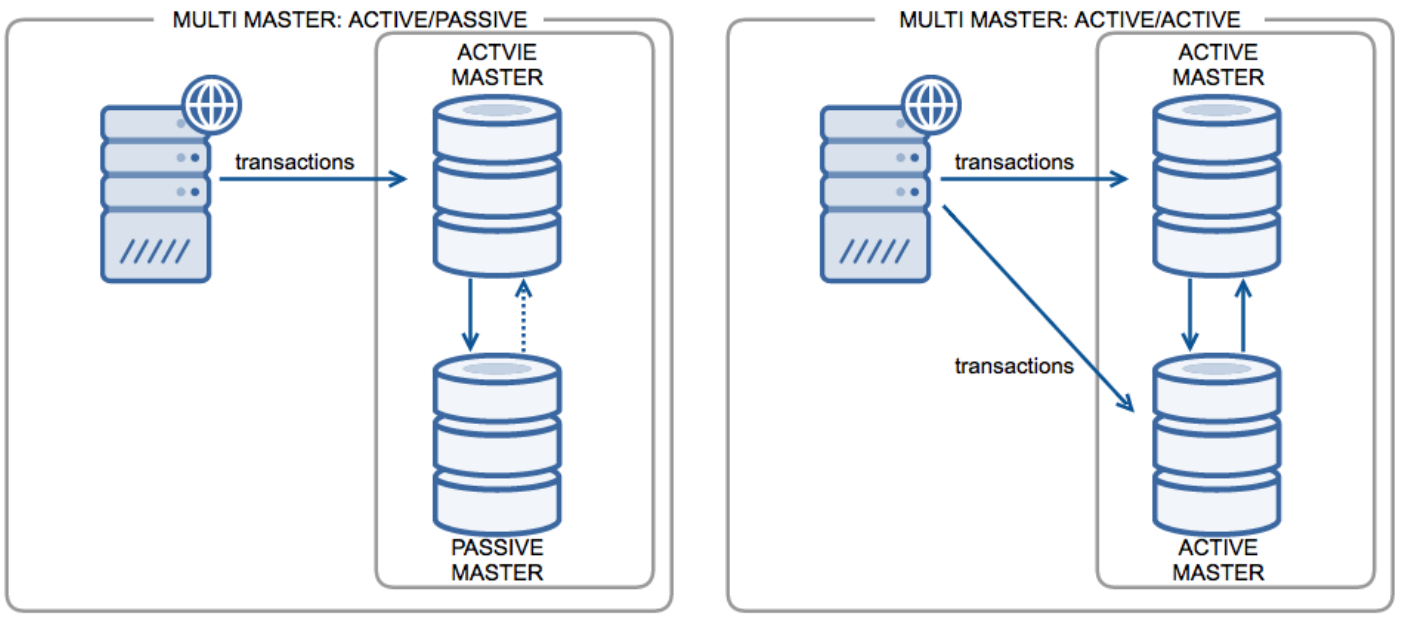
XA transactions never became popular because of three primary reasons:

- **Rarely Needed:** Each database acting as a XA resource was a monolithic ACID compliant database which usually stored most of a given application's data. The need for ACID across such independent databases usually involved integrating disparate applications altogether. Developers rightly concluded that its simpler to solve this problem at the app layer using web services.
- **High Latency & Low Throughput:** Implementation of the XA 2PC protocol relied on full row-level locks. Result was that locks were held for a long time than necessary, leading to poor performance in terms of high latency and low throughput.
- **Inconsistent & Buggy Implementations:** Every database and app server vendor implemented the XA specification as it saw fit and without thorough testing of failure conditions. Its not surprising that there were inconsistencies and bugs across such implementations and users could not make XA transactions work as seamlessly as expected.

The good news is that the issues outlined above are not applicable in today's world. Transactional data can no longer fit in a monolithic database server resulting in a higher-than-ever-before need for transactions on distributed databases. Pessimistic row-level locks have been replaced with optimistic, column-level locks that use automatic conflict detection to ensure low latency and high throughput. And finally, the same protocol runs on every node of the database cluster removing the issue of inconsistent implementations on top of an abstract standard.

## 4. Multi-Master is Same as Distributed ACID

All monolithic SQL databases (including Amazon Aurora) and even a few proprietary NoSQL services (such as Amazon DynamoDB and Azure Cosmos DB) support multi-master replication especially in the context of a multi-region deployment. The replication can be configured to be active/passive , where only one node is writable and the other nodes are a hot standby. Another configuration is active/active, where all master nodes are writable.




*Source: SeveralNines Blog*

**A common misunderstanding with active/active multi-master is that the entire cluster is now globally consistent and distributed ACID compliant.** In a multi-master system, each master node contains the entire dataset and hence handles writes for all the data without any need for distributed transactions. Changes on one master are replicated to other masters asynchronously for read scaling and disaster recovery with write scaling remaining an unsolved problem. In active/active, the benefit of write scaling is negated by conflict-ridden concurrent writes on same rows at different masters. Such

conflicts are resolved without explicit knowledge of the app using non-deterministic heuristics such as Last-Write-Wins (LWW) and Conflict-Free Replicated Data Types (CRDT). Net result is customer data loss and poor engineering productivity when compared to fully ACID-compliant distributed databases.

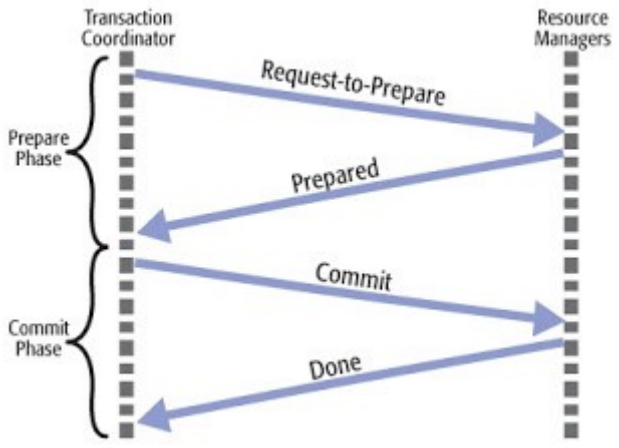
## 5. Application Level ACID is Easy to Implement

Developers sometimes attempt to implement ACID transactions in their application logic. The most basic option is single-operation, single-row ACID on an eventually consistent database as documented by AWS DynamoDB. A more advanced option is fully distributed ACID on a strongly consistent database such as MongoDB. This section explains why neither of these is a good idea given the effort needed to get this right.

A Primer to ACID Transactions details what's needed for ACID transactions when only one database shard is involved. On the other hand, the implementation for distributed transactions is a lot more complex given the need to manage updates across multiple shards located on multiple nodes.
A transaction manager is usually needed to act as the coordinator. It can be either a system independent of the database nodes or can be built directly into the database nodes. Let's look at how it works to achieve the ACID guarantees.

### Achieving Atomicity

The transaction manager needs a mechanism to track the start, progress and end of every transaction along with the ability to make provisional updates across multiple nodes in some temporary space. Conflict detection, rollbacks, commits and space cleanups are also needed. Using Two-phase commit (2PC) protocol or one of its variations is the most common way to achieve atomicity.




### Achieving Consistency

The database's replication architecture has to be strongly consistent using a formally proven distributed consensus protocol such as Paxos or Raft. The transaction manager will rely on the correctness of a single operation on a single row to enforce the broader ACID-level consistency of multiple operations over multiple rows. BASE systems such as Apache Cassandra that use quorum reads and quorum writes as tunable consistency controls on top of eventually consistent replication cannot provide true strong consistency. Given the absence of rollbacks after failures, dirty reads from failed writes and deleted data resurfacing are common in such systems.

### Achieving Isolation

The transaction manager cannot simply stamp all the operations inside a single transaction with its own time—other instances of the transaction manager on other nodes can operate on the same data but won't have the same view of the time. Getting even a partial ordering of the operations in a transaction becomes a challenge. Protocols such as Network Time Protocol (NTP) allow the times at each node to be synchronized over the public internet with a common source of truth but there's still no guarantee that all nodes will see the exact same time since internet network latency is unpredictable. In reality, nodes exhibit clock skew/drift even with NTP turned on. So a more sophisticated time tracking mechanism is needed.

### Achieving Durability

Committed transactions should be stored in a disk-based persistent device where their results are preserved across system restarts and node failures. Since disks themselves can fail, using replication to store copies of the same data on other nodes is a must-have.

## 6. Distributed Transactions = Poor Performance

So, can distributed databases support ACID transactions without the performance issues of the past? The answer is Yes. We are now in the renaissance era of modern distributed databases that come with built-in support for all the three types of ACID. Examples in the distributed SQL category are Google Cloud Spanner, YugaByte DB, CockroachDB and TiDB and in the NoSQL category are FoundationDB and FaunaDB. For example, Yes We Can! Distributed ACID Transactions with High Performance details the various optimizations YugaByte DB uses to deliver low latency and high throughput transactions.

## Summary

The table below summarizes the ACID spectrum in today's distributed databases. As we can see, the original NoSQL design paradigm of forsaking strong consistency and ACID is increasingly becoming irrelevant.

| ACID | Consistency | NoSQL | Distributed SQL |
|---|---|---|---|
| NONE<br>↓<br>ACID SPECTRUM<br>↓<br>FULL | | | |
| ✗ | Eventual Consistency | Amazon DynamoDB, Apache Cassandra, Couchbase, ScyllaDB, ArangoDB, Aerospike | ✗ |
| Single Row | Strong Consistency | Azure Cosmos DB (Single Operation in a Single Region) | YugaByte DB, Google Cloud Spanner, CockroachDB, TiDB |
| Single Shard | | MongoDB | |
| Distributed (Cross-Shard) | | FoundationDB, FaunaDB | |

At the same time, new research from Google (such as the Spanner and Percolator papers) and academia (such as Yale's Calvin paper) has served as inspiration for a new generation of distributed ACID compliant databases. These databases are also aiming to solve the performance problems observed in previous implementations such as XA. We review some of the issues involved in our post Implementing Distributed Transactions the Google Way: Percolator vs. Spanner.

JULY 22, 2018

ACID TRANSACTIONS , NOSQL , SQL

Sid Choudhury | VP, Product

July 22, 2018

## Related Posts