Macrometa’s approach to solving complex, geo-distributed data challenges

Introduction

The last 15 years have seen several successive waves of big data platforms and companies offering new database and analytics capabilities. Leveraging the public cloud, these large-scale distributed data systems work well in centralized, single region, or single data center topologies and are geared for "**Read intensive**" data problems.

Companies such as Cloudera, Snowflake, and, more recently, Databricks are examples of successful companies that have innovated with novel "read architectures" that reduce the structural cost of reading immutable data and built new big data-orientated analytics apps and capabilities by exploiting their lower costs of reads.

We are now faced with new challenges with workloads and use cases that are "**write intensive**". The write path is far more challenging than the read path. This is because the read path is built on immutable or non-changing data. The write path necessarily involves mutable data with varying rates of change and growth in the data sets of use cases.

Additionally, the write path today has also evolved from the client/server's "request – response" based interaction model to new, streaming data architectures where streams of "events" need to be processed as they are created or generated by various sources.

A similar cascade of innovations is needed on the write path to solve the structural costs of data writes as big data platforms have done to change the structural costs of data reads.
We summarize the challenges of the write path in modern cloud data platforms as follows:

1. *Expensive Writes* - (Ex: DynamoDB - writes are 16x to 64x more expensive than reads).
2. *Centralized architectures* - Introduces large access latencies for globally distributed users and as well as large transit latencies and network transfer costs for data to be shipped to the cloud.
3. *Require network communication-intensive and coordinated approaches* to consistency such as using state machine replication or consensus and therefore are only feasible within a single cloud region (due to the need for ultra-low latency and reliable data center-class networks).
4. *Require synchronous replication* to ensure all nodes in a distributed system make forward progress on writes with transaction semantics where either all related writes are accepted or all are rejected. In combination with point #3 above, replication and coordination end up being the Achilles heel for write-intensive distributed systems.

Today's enterprise apps and cloud workloads are no longer "read intensive" but are a combination of:

1. "**Write intensive**" (ex: IoT, Monitoring, ClickStreams, Fraud Detection, etc.) or
2. "**Read-Write balanced**" (ex: E-Commerce, Gaming, Adtech, etc.)

This is because modern cloud-native architectures don't use just one database behind them for persistence but instead use several specialized databases and data stores for different types of semi-structured and unstructured data. This "*polyglot persistence*" pattern used by modern apps further exacerbates the write path and write intensity problem as now a single application-level write generates a successive cascade of amplified writes to multiple datastores underneath the application.

The "**Multi database/datastore mutations**" pattern where an application-level write results in several cascading updates to multiple independent databases or datastores causes very high levels of write amplification and resulting costs.

Attempts to solve these problems using conventional techniques of "*write back*" and "*write through*" caching no longer work for such polyglot backend patterns – one simply cannot buffer writes in a single buffer and fan out to multiple databases and data stores underneath without completely giving up on consistency and transaction semantics.

For use cases where multi-region data or edge-based data processing are needed, the problems are further exacerbated by the physical link latency, topology, and reliability of the wide-area network. These use cases have to contend with well-known issues such as:

1. Network latencies with 100s of ms,
2. Unreliable & jittery networks.

Given the challenges just described current centralized distributed data systems and the technologies that underpin them cannot be generalized well to fit edge & multi-cloud workloads.

Macrometa has been now for several years, focused on solving the challenges of the write path. We don't just want to solve the cost and consistency problem of the write path but also enable globally distributed stateful apps & web services that can run in 10s and 100s of regions worldwide concurrently with less than 50 ms end to end latency for data access by 95% of the world's population.

This is a nontrivial computer science problem because:

1. It requires our platform to run in wide-area deployments where nodes forming the database may be separated by more than 100ms of latency and unreliability in the network connections.
2. It requires our platform to run across 10s to 100s of data centers and present a single system image (SSI).
3. It requires us to have a significantly better write cost structure and cost-performance ratio than current cloud data systems built on centralized architectures.

Macrometa accomplishes this with a novel architecture that combines geo-distributed, coordination-free write and replication techniques and a multi-modal data platform with tunable consistency levels to solve these problems in an edge-native way.

The approach for cheaper writes

In current centralized cloud data architectures, writes are expensive due to the sheer number of writes an application generates that result in 2 levels of amplification:

1. Writes are amplified as they fan in the polyglot persistence pattern and
2. Further amplified by multiple inline updates to the underlying indexes and associated metadata structures in the persistence layer for each underlying database or datastore.

Macrometa solves this via a combination of two approaches:

1. A single copy of data with multiple data models and a unified noSQL query. Instead of having different databases for every type of unstructured or semi-structured data, Macrometa keeps a single copy of data that may be read and mutated by different query interfaces (K/V, docs, graphs, streams, etc.). This enables the use and benefit of polyglot persistence patterns without the Multi database/datastore mutations' amplification cost.
2. Using append-only logs to process writes and generates in-memory materialized views in real-time for each mutation. Changes to the views are transformed into CRDT operations and shipped with causal ordering to the peers. This considerably reduces the cost of writes for Macrometa within a region.

The approach to a Coordination Free Architecture

Today's cloud databases and distributed data platforms are built on coordination-based architectures. They rely on state machine replication and consensus protocols which are highly chatty & synchronous in nature.

Coordination creates three primary penalties for distributed systems –

1. Increased Latency (due to stalled execution)
2. Decreased Throughput
3. Unavailability

For example, if a transaction takes “d” seconds to execute, the maximum throughput of the coordinating transactions operating on the same data items is 1/d. *Source:* *Coordination Avoidance in Database Systems (vldb.org)*

This is typically not an issue if the database is within a single availability zone or a single data center. There the network delays are small (i.e., hundreds of microseconds or low single-digit milliseconds), thereby permitting from few 100s to few 1000s of coordinating transactions per item per second. However, if the database is spread across multiple regions/PoPs, the costs will dramatically increase. The delays are lower bound by the network latencies, which run into 100s of milliseconds, as shown in the below picture.

	H2	H3
H1	0.55	0.56
H2		0.50

(a) Within us-east-b availability zone

	C	D
B	1.08	3.12
C		3.57

(b) Across us-east availability zones

	OR	VA	TO	IR	SY	SP	SI
CA	22.5	84.5	143.7	169.8	179.1	185.9	186.9
OR		82.9	135.1	170.6	200.6	207.8	234.4
VA			202.4	107.9	265.6	163.4	253.5
TO				278.3	144.2	301.4	90.6
IR					346.2	239.8	234.1
SY						333.6	243.1
SP							362.8

(c) Cross-region (CA: California, OR: Oregon, VA: Virginia, TO: Tokyo, IR: Ireland, SY: Sydney, SP: São Paulo, SI: Singapore)

Table 2.1: Mean RTT times on EC2 (min and max highlighted)

Source: Highly Available Transactions: Virtues and Limitations (Extended Version) (arxiv.org)

This, in turn, leads to a dramatic reduction in throughput. For example, 2 to 3 coordinating transactions per second within an 8-region cluster. This throughput is too low and unfeasible for write-intensive or read-write balanced workloads and use cases.

On average, intra-datacenter communication is

1. 1.82 to 6.38 times faster than geographically co-located data centers and
2. 40 to 647 times faster than geographically distributed datacenters.

Source: Highly Available Transactions: Virtues and Limitations (Extended Version) (arxiv.org)

Macrometa eschews the use of coordination in its architecture and instead utilizes Conflict-Free Replicated Data Types (CRDTs) to achieve coordination-free architecture and sidesteps the problems created by coordination-based systems (described above).

A CRDT is an abstract data type with a well-defined interface, designed to be concurrently modified by multiple, independent processes or replicated to be modified by multiple independent processes. CRDTs exhibit the following important properties:

1. Any replica can be modified without coordinating with another replicas;
2. When any two replicas have received the same set of updates, they reach the same state, deterministically, by adopting mathematically sound rules to guarantee state convergence. *Source: Conflict-free Replicated Data Types (CRDTs) (arxiv.org)*

CRDTs have three fundamental properties that enable coordination-free synchronization - Commutativity, Associativity & Idempotence. These properties enable a system to replicate in a coordination-free manner while converging to the same state independently and deterministically.

In Macrometa, mutation operations to the database like inserts, updates, and deletes are transformed into particular CRDT operations and shipped to peers as a partially ordered log of operations on reliable geo-replicated streams.

To solve the issues with geo-distributed time stamping (synchronizing clocks over the WAN is another well-known problem), Macrometa attaches an optimized variant of vector clocks called "Version Vectors" to each message broadcast as a logical timestamp and uses the causality information in the vector clock to decide at each destination how a message should be processed.

Note: Macrometa transforms and ships the operations (using Operations-Based CRDTs) and not the system's delta state (like State-Based CRDTs).

This coordination-free architecture enables the Macrometa platform to sidestep the three penalties introduced by current coordination-based architectures, i.e., increased latency (due to stalled execution), decreased throughput & unavailability.

Solving the challenge of unbounded log growth

One common challenge with CRDT systems is that the nodes (i.e., PoPs) have to keep the entire history of operations so that new nodes/PoPs may join and converge. This increases the storage cost as the history of operations can grow in an unbounded fashion. Getting around this side effect of CRDTs and logs requires coordination between nodes to pick a causally stable checkpoint and truncate the log at the said checkpoint.

In concurrent data types, timestamps' order information may only be needed for an operation as long as its concurrent operations are delivered or expected. However, this op log information is useless once no concurrent operations are expected for a given operation. This operation can then be called a "causally stable" operation, and thus, it makes sense to get rid of this extra meta-data. Source: Pure Operation-Based Replicated Data Types (arxiv.org)

Given our philosophy of being completely coordination-free, Macrometa implements a novel coordination-free approach to creating causally stable checkpoints in the log without needing nodes to synchronize metadata about causal stability across regions/PoPs.

Our co-ordination free approach to log truncation determines a set of causally stable operations, and garbage collects the extraneous operations and vector clock metadata in the log at each PoP. This

enables Macrometa to substantially reduce the storage costs and associated log traversal IO costs (again). This also enables Macrometa to scale to a large number of PoPs (100s of regions).

Tunable Consistency Levels

Tunable consistency levels enable developers to granularly choose the best consistency and performance for their data at a table/collection level. Many of the hardest challenges of distributed data platforms involve consistency guarantees.

Geo-replicated, distributed data platforms that support complex online applications, such as e-commerce, etc., must provide an "always-on" experience where operations always complete with low latency. Today's systems often sacrifice consistency to achieve these goals, exposing inconsistencies to their clients and necessitating complex application logic.

For example, AWS DynamoDB, Cassandra, LinkedIn Voldemort, etc., provide ***Eventual Consistency*** to achieve Availability & Partition Tolerance. Eventual Consistency ensures that writes to one data center will eventually appear at other data centers, and if all data centers have received the same set of writes, they will have the same values for all data.

The problem with eventual consistency is that it does not say anything about the ***ordering of operations***. This means that different data centers can reflect arbitrarily different sets of operations. For example, if someone connected to the West Coast data center sets A=1, B=2, and C=3, then someone else connected to the East Coast data center may see only B=2 (not A=1 or C=3), and someone else connected to the European data center may see only C=3 (not A=1 or B=2).

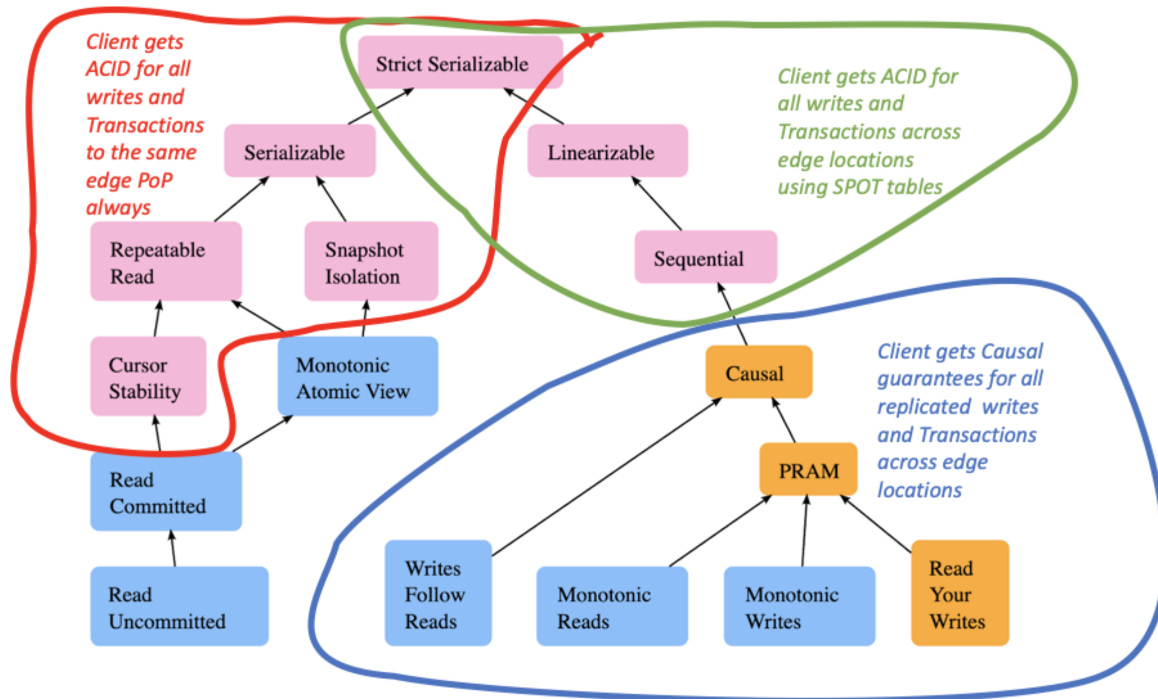
This makes programming with eventually consistent systems very hard and opens up many corner cases where the application or the data platform do not know how to solve for operations that appear out of order. The out-of-order arrival leads to many serious anomalies in eventually consistent systems.

We recommend a reading of Don't Settle for Eventual: Scalable Causal Consistency for Wide Area Storage with COPS (cornell.edu) for a thorough discussion of these anomalies.

Macrometa platform provides tunable consistency levels that are stronger than Eventual Consistency seen in current distributed databases:

1. Strict Serializability within a region (default)
2. Casual+ consistency across regions (default) and
3. Strict Serializability across regions at a collection level.

Strict Serializability provides ACID semantics within a region. Macrometa platform uses MVCC to provide snapshot isolation. With snapshot isolation, a transaction observes a state of the data as when the transaction started. Read and write transactions are thus isolated from each other without any need for locking. Each mutation generates new versions of a document with automatic garbage collection.



Source: [Consistency Models (jepsen.io)]

Macrometa platform provides Casual+ (Causal plus) consistency across regions. This automatically covers the following consistency guarantees:

- **Causal consistency**. If process A has communicated to process B that it has updated a data item, subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C with no causal relationship to process A is subject to the normal eventual consistency rules.
- **Read-your-writes consistency**. This is an important model where process A has updated a data item, always accesses the updated value, and will never see an older value. This is a special case of the causal consistency model.
- **Session consistency**. This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created, and the guarantees do not overlap the sessions.
- **Monotonic read consistency**. If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.
- **Monotonic writes consistency**. In this case, the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

Source: [Eventually Consistent - Revisited - All Things Distributed]

Macrometa platform also provides Strict serializability across regions at a collection level for use cases that require it. For example, in use cases related to account withdrawals (in banking applications),

concurrent coordination free operations can result in undesirable outcomes like negative account balances. This is typically called the "double-spend problem".

To ensure correct behavior, the database system must coordinate the execution of these operations across all regions the data resides. Macrometa platform enables users to choose collections that should have strict consistency to address these types of use cases. Macrometa then behaves as a "CP" system for these operations here at the expense of latency & availability.

Typically, in most applications, 95% of the workflows are satisfied by Strict serializability within a region or by Causal consistency across regions. These don't require strict serialization across the globe.

So, by utilizing the flexible consistency levels model, Macrometa platform users can make better trade-offs between latency, throughput, availability & consistency using global strict consistency only where needed.

Polyglot multi-modal data persistence patterns

Today when one chooses a database, they are choosing three things –

1. Storage Technology
2. Data Model and
3. Query/API Language.

For example,

- If you choose AWS DynamoDB, you are choosing the dynamo dB storage engine, its columnar data model, and the dynamo query language.
- Similarly, if you choose MongoDB, you are choosing the MongoDB storage engine, a document data model, and the MongoDB API.
- Similarly, for Neo4j, i.e., you are choosing its storage engine, a graph data model, and the Cypher Query language.

All these databases need the same set of features and are tightly coupled between all of the layers. For example, all these systems provide indexes, and the notion of an index exists in all three layers. This commonality extends to some extent to messaging systems as well, like Kafka where you are, in essence, choosing a storage technology (aka append-only logs), data model (aka messages), and query language (aka pub-sub APIs).

Key-Value databases, Document databases, Graph databases, Streams, etc., all make sense in the right context, and quite often, different parts of an application call for different choices. This creates a tough decision:

- Use a whole new database to support a new data model, or
- Try to shoehorn data into your existing database.

Using a whole new database approach leads to making N copies of data in different formats and processing multiple times. This also forces developers to expend a substantial portion of time and effort in writing substantial integration glue code. On the other hand, shoehorning data into an existing database leads to unnecessary complexity and lower performance.

Macrometa platform takes a layered approach and decouples these three layers:

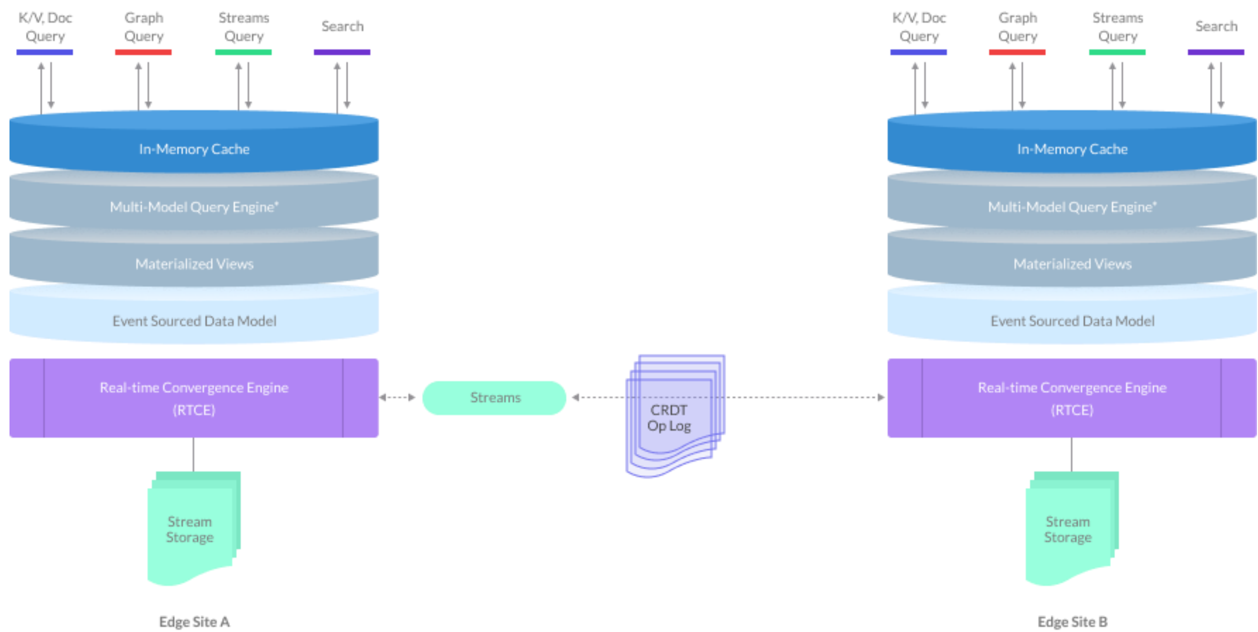
- For the storage layer, it uses Append-only logs & Log-Structured Merge trees.
- For the data model layer, it adds rich data models like Key-Value, Docs, Graphs, Search & Streams on top of the storage layer.
- For the *Query/API layer*, it layers multiple query engines like C8QL, Dynamo Mode, Redis. Mode* & Mongo Modes* on top of the data model layer. Similarly, it layers Apache Kafka, Apache Pulsar & AWS Kinesis* protocols for data-in-motion.

Being an edge platform with the ability to connect and cache data from centralized databases also means that Macrometa must support a heterogeneous mix of noSQL interfaces on its own but in a way that does not create additional storage and IO overhead by keeping copies of data for each specific storage format (K/V, docs, graphs, streams, etc.).

Our approach's salience is that it enables our data platform to provide multiple data models on a single copy of data reducing storage & processing costs while still supporting the polyglot persistence pattern, increasing performance and developer velocity for both data-in-motion and data-at-rest use cases.

Architecture

Macrometa architecture brings together all the above aspects to provide a geo-distributed coordination-free multi-modal data platform, as shown in the below image. User mutations are transformed into CRDT ops, tagged with version vectors, and shared with peer nodes (PoPs) via geo-replicated causal ordered reliable streams utilizing publish-subscribe semantics.



Incoming CRDT ops are processed by our Real-Time convergence engine (RICE) to update materialized views in real-time and stored in event source storage (aka storage layer). These views form the data model layer providing a common abstraction to support polyglot data model patterns, i.e., key-value, doc, graphs, search & streams.

The query/API layer builds on top of the data model layer and provides multiple query engines like C8QL (a unified query language for KV, Docs, Graphs & Search) as well as wire compatible interfaces like Dynamo mode, Redis mode* & Mongo mode*. Similarly, for the streams, the query/API layer provides native support for Apache Kafka protocol, Apache Pulsar protocol, and AWS Kinesis mode*.

Edge & multi-cloud environments require geo-distributed data platforms that handle natively the constraints imposed by these environments. Bottom line, do not use a hammer to chop the wood!