Dynamo at 15

What Worked and What's Next in Majority-Quorum Databases

C. Scott Andreas, Apple Inc.
HPTS 2022 – Pacific Grove, California

Outline

- A look back at Dynamo ca. 2007
- Apache Cassandra's journey as a Dynamo-derived database
  - What stood the test of time?
  - What needed revisiting?
- What's new and what's next in majority-quorum databases?

Apache Cassandra

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service

Mostafa Elhemali, Niall Gallagher, Nicholas Gordon, Joseph Idziorek, Richard Krog
Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somu Perianayagam ,Tim Rath
Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, Akshat Vig
dynamodb-paper@amazon.com
Amazon Web Services

Abstract

Amazon DynamoDB is a NoSQL cloud database service that provides consistent performance at any scale. Hundreds of thousands of customers rely on DynamoDB for its fundamental properties: consistent performance, availability, durability, and a fully managed serverless experience. In 2021, during the 66-hour Amazon Prime Day shopping event, Amazon systems - including Alexa, the Amazon.com sites, and Amazon fulfillment centers, made trillions of API calls to DynamoDB, peaking at 89.2 million requests per second, while experiencing high availability with single-digit millisecond performance. Since the launch of DynamoDB in 2012, its design and implementation have evolved in response to our experiences operating it. The system has successfully dealt with issues related to fairness, traffic imbalance across partitions, monitoring, and automated system operations without impacting availability or performance. Reliability is essential, as even the slightest disruption can significantly impact customers. This paper presents our experience operating DynamoDB at a massive scale and how the architecture continues to evolve to meet the ever-increasing demands of customer workloads.

1 Introduction

Amazon DynamoDB is a NoSQL cloud database service that supports fast and predictable performance at any scale. DynamoDB is a foundational AWS service that serves hundreds of thousands of customers using a massive number of servers located in data centers around the world. DynamoDB powers multiple high-traffic Amazon properties and systems including Alexa, the Amazon.com sites, and all Amazon fulfillment centers. Moreover, many AWS services such as AWS Lambda, AWS Lake Formation, and Amazon SageMaker are built on DynamoDB, as well as hundreds of thousands of customer applications.

These applications and services have demanding operational requirements with respect to performance, reliability, durability, efficiency, and scale. The users of DynamoDB rely on its ability to serve requests with consistent low latency. For DynamoDB customers, consistent performance at any scale is often more important than median request service times because unexpectedly high latency requests can amplify through higher layers of applications that depend on DynamoDB and lead to a bad customer experience. The goal of the design of DynamoDB is to complete *all* requests with low single-digit millisecond latencies. In addition, the large and diverse set of customers who use DynamoDB rely on an ever-expanding feature set as shown in Figure 1. As DynamoDB has evolved over the last ten years, a key challenge has been adding features without impacting operational requirements. To benefit customers and application developers, DynamoDB uniquely integrates the following six fundamental system properties:

DynamoDB is a fully managed cloud service. Using the DynamoDB API, applications create tables and read and write data without regard for where those tables are stored or how they're managed. DynamoDB frees developers from the burden of patching software, managing hardware, configuring a distributed database cluster, and managing ongoing cluster operations. DynamoDB handles resource provisioning, automatically recovers from failures, encrypts data, manages software upgrades, performs backups, and accomplishes other tasks required of a fully-managed service.

DynamoDB employs a multi-tenant architecture. DynamoDB stores data from different customers on the same physical machines to ensure high utilization of resources, enabling us to pass the cost savings to our customers. Resource reservations, tight provisioning, and monitored usage provide isolation between the workloads of co-resident tables.

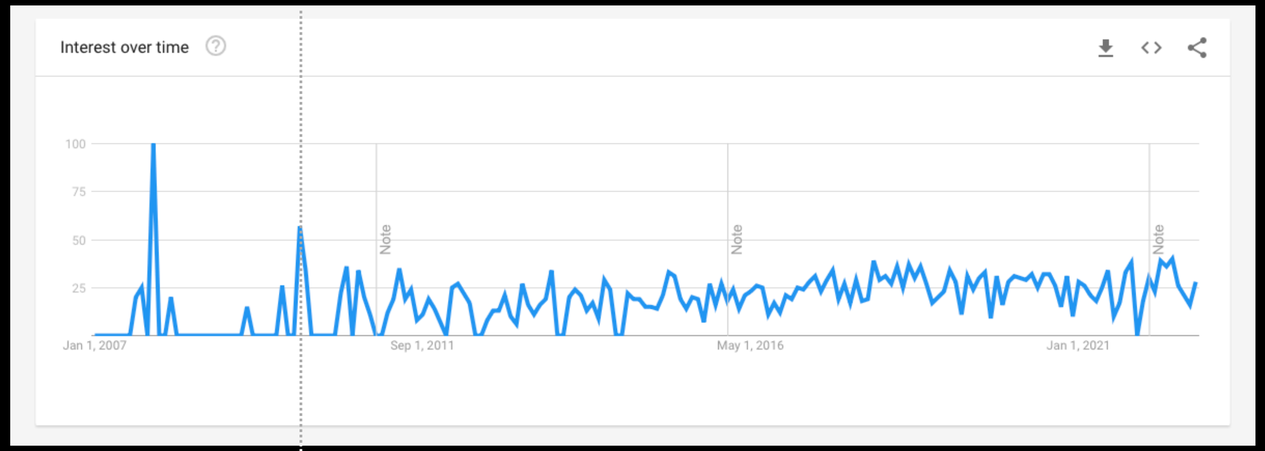
DynamoDB achieves boundless scale for tables. There are no predefined limits for the amount of data each table can store. Tables grow elastically to meet the demand of the customers' applications. DynamoDB is designed to scale the resources dedicated to a table from several servers to many thousands as needed. DynamoDB spreads an application's data across more servers as the amount of data storage and the demand for throughput requirements grow.

DynamoDB provides predictable performance. The simple

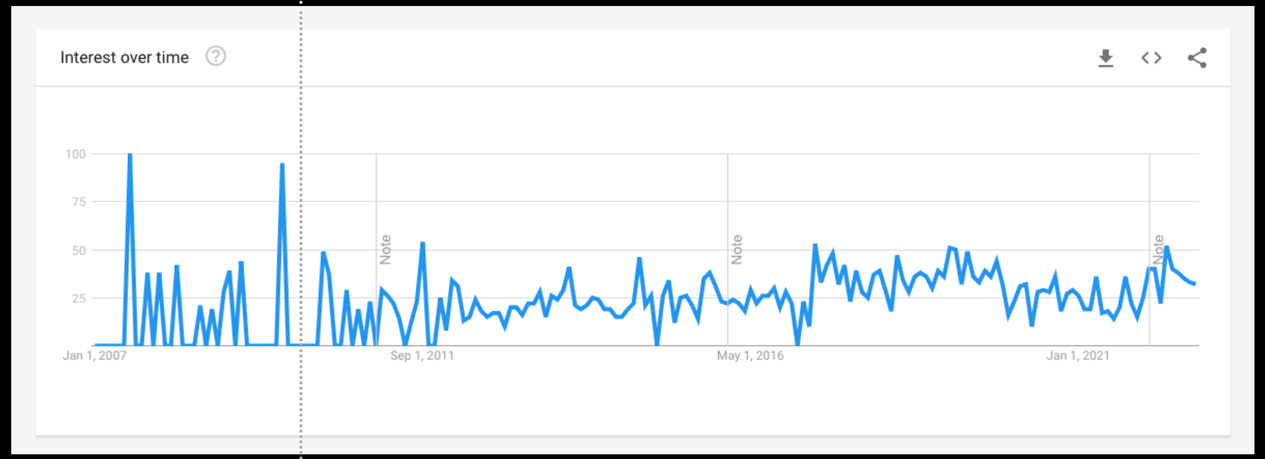
Landscape in 2007

Hardware	Operating Systems	Databases
Intel Penryn (45nm)	**Linux 2.6.2**	**MySQL 5**
First 1TB HDD (250GB common)	**Windows Vista**	**SQL Server 9**
~32-64 GB Max RAM (Major manufacturers)	**Mac OS X 10.5**	**Postgres 8.2**
SATA 2.6 Slim Connector	**Solaris 10**	**Oracle 11g**

Eventual Consistency



Consistent Hashing



November 2007

Google Trends - “Eventual Consistency” and “Consistent Hashing,” accessed Sept 30, 2022.

“A Highly Available Key-Value Store”

Prioritizing …

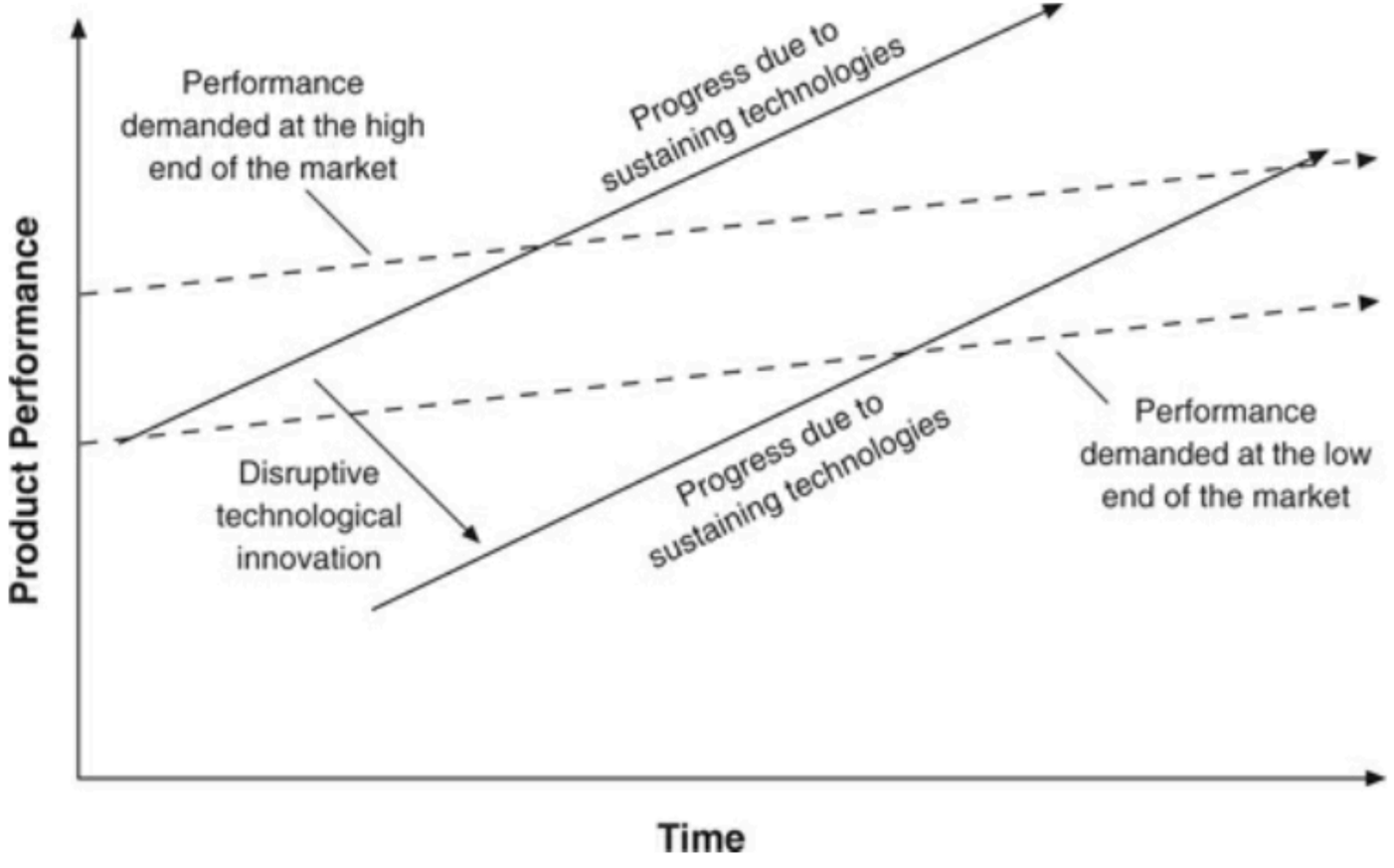
- Leaderless architecture
- Incremental scalability
- API: `get(key)`, `put(key, context, bytes)`
- Availability as paramount (99.999%+)
- Aggressive Latency SLA (focus: 99.9th percentile)

“A Highly Available Key-Value Store”
Without …

- Query Language
- Query Planner / Optimizer
- Transactions
- Indexes
- MVCC / Snapshot Isolation
- Rich Data Types
- Aggregations
- Views
- Functions / Procedures
- Storage engine (pluggable)

What is a database?

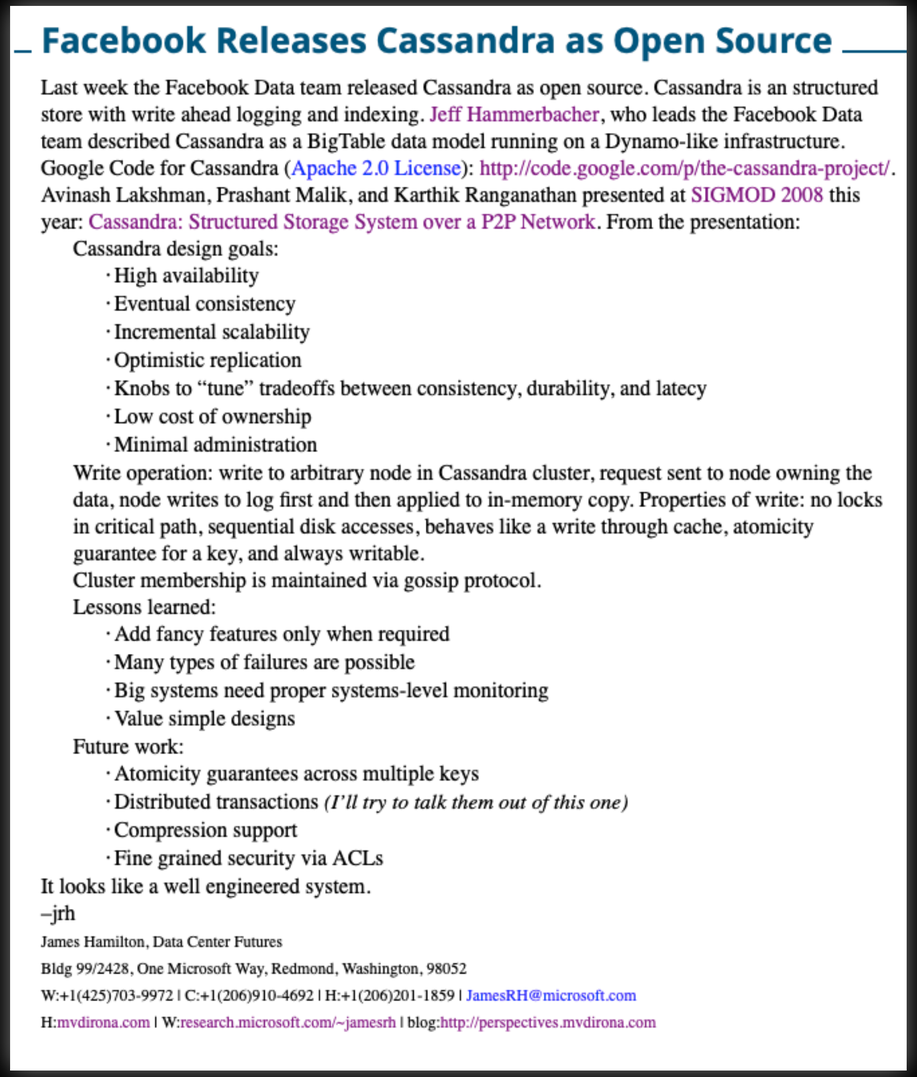
Figure I.1 The Impact of Sustaining and Disruptive Technological Change



The Innovator's Dilemma - Clayton M. Christensen. Harvard Business Review Press, 1997 - p. 16

Apache Cassandra
As Introduced

- First release coauthored by coauthor of Dynamo paper, then at Facebook. LSM storage engine engine, majority-quorum architecture.
- OSS'd in 2008, ASF incubation 2009, top-level project 2010.
- Dynamo Concepts - Huge overlap:
  Consistent hashing / DHT, tunable consistency, gossip, hinted handoff, merkle trees, timestamp conflict resolution.
- James Hamilton: "It looks like a well engineered system."



Facebook Releases Cassandra as Open Source

Last week the Facebook Data team released Cassandra as open source. Cassandra is an structured store with write ahead logging and indexing. Jeff Hammerbacher, who leads the Facebook Data team described Cassandra as a BigTable data model running on a Dynamo-like infrastructure. Google Code for Cassandra (Apache 2.0 License): http://code.google.com/p/the-cassandra-project/. Avinash Lakshman, Prashant Malik, and Karthik Ranganathan presented at SIGMOD 2008 this year: Cassandra: Structured Storage System over a P2P Network. From the presentation:

Cassandra design goals:

- High availability
- Eventual consistency
- Incremental scalability
- Optimistic replication
- Knobs to "tune" tradeoffs between consistency, durability, and latecy
- Low cost of ownership
- Minimal administration

Write operation: write to arbitrary node in Cassandra cluster, request sent to node owning the data, node writes to log first and then applied to in-memory copy. Properties of write: no locks in critical path, sequential disk accesses, behaves like a write through cache, atomicity guarantee for a key, and always writable.

Cluster membership is maintained via gossip protocol.

Lessons learned:

- Add fancy features only when required
- Many types of failures are possible
- Big systems need proper systems-level monitoring
- Value simple designs

Future work:

- Atomicity guarantees across multiple keys
- Distributed transactions *(I'll try to talk them out of this one)*
- Compression support
- Fine grained security via ACLs

It looks like a well engineered system.

–jrh

James Hamilton, Data Center Futures
Bldg 99/2428, One Microsoft Way, Redmond, Washington, 98052
W:+1(425)703-9972 | C:+1(206)910-4692 | H:+1(206)201-1859 | JamesRH@microsoft.com
H:mvdirona.com | W:research.microsoft.com/~jamesrh | blog:http://perspectives.mvdirona.com

"Facebook Releases Cassandra as Open Source" - James Hamilton, *Perspectives. July 2008.*

Apache Cassandra

What Problem Are We Solving?

- **Scale:** 3 to 1500+ database instances.
- **Availability:** 99.999% common, 99.9999% achievable.
- **Distribution:** Active-Active across up to five regions.
- **Density:** Databases up to ~2.5 PiB in size.
- **Velocity:** Millions of queries per second.
- **Capability:** Strong consistency, linearizable transactions.

Apache Cassandra

What Worked Great

- **Leaderless Architecture:** No distinguished nodes.
  - Equal latency across regions (no “primary”).
  - Faults localized to replica/replica set rather than global.
  - Avoids bottlenecks, gray-mode failures, election complications.
- **Majority Quorum Design:**
  - P(Failure): Concurrent loss of replicas across failure domains.
  - Enables striping replicas across rack / network / power domains.
  - Fault domain-aware planning enables 99.9999% availability.
  - Strong consistency via overlapping quorums, blocking read repair.

Apache Cassandra

What Worked Great

- **Leaderless Architecture:** No distinguished nodes.
  - Equal latency across regions (no “primary”).
  - Faults localized to replica/replica set rather than global.
  - Avoids bottlenecks, gray-mode failures, election complications.
- **Majority Quorum Design:**
  - P(Failure): Concurrent loss of replicas across failure domains.
  - Enables striping replicas across rack / network / power domains.
  - Fault domain-aware planning enables 99.9999% availability.
  - Strong consistency via overlapping quorums, blocking read repair.

Apache Cassandra

Becoming a Database

User-Facing Features

CQL Query Language
Secondary Indexes

Strong Consistency

Single-Key
Transactions

New Storage Engine
UDFs / UDAFs

Spark / HTAP Integration
Distributed Transactions

Becoming a Database

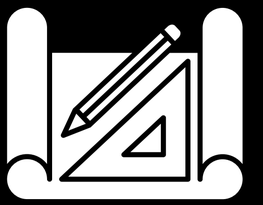
Foundational Work: 2018 - 2022

- **Quality Gap:** “EC k/v” roots to “SC database” - new expectations.
  - **Property-Based Testing:** Randomized input, validation by model checker.
  - **Simulation:** Deterministic execution via managed executors / mutexes.
  - **Upgrade Testing:** Automating clone, upgrade, exhaustive validation.
- **Rethinking Deprecation:** Lose features, lose users.
- **Rethinking Dynamo Concepts:** What do we need to become?

Rethinking Dynamo Concepts

Forging New Paths

- **Gossip:** Membership and ownership as fundamentally transactional concerns.
- **Eventual Consistency:** Linearizability is tablestakes for modern applications.
- **Anti-Entropy:** Reducing overhead via immutability, invariants.
- **Replication:** Decoupling quorum size from replication via Witness Replicas.
- **Smart-Client Routing:** Limitations at 100k+ clients.



Distributed Transactions

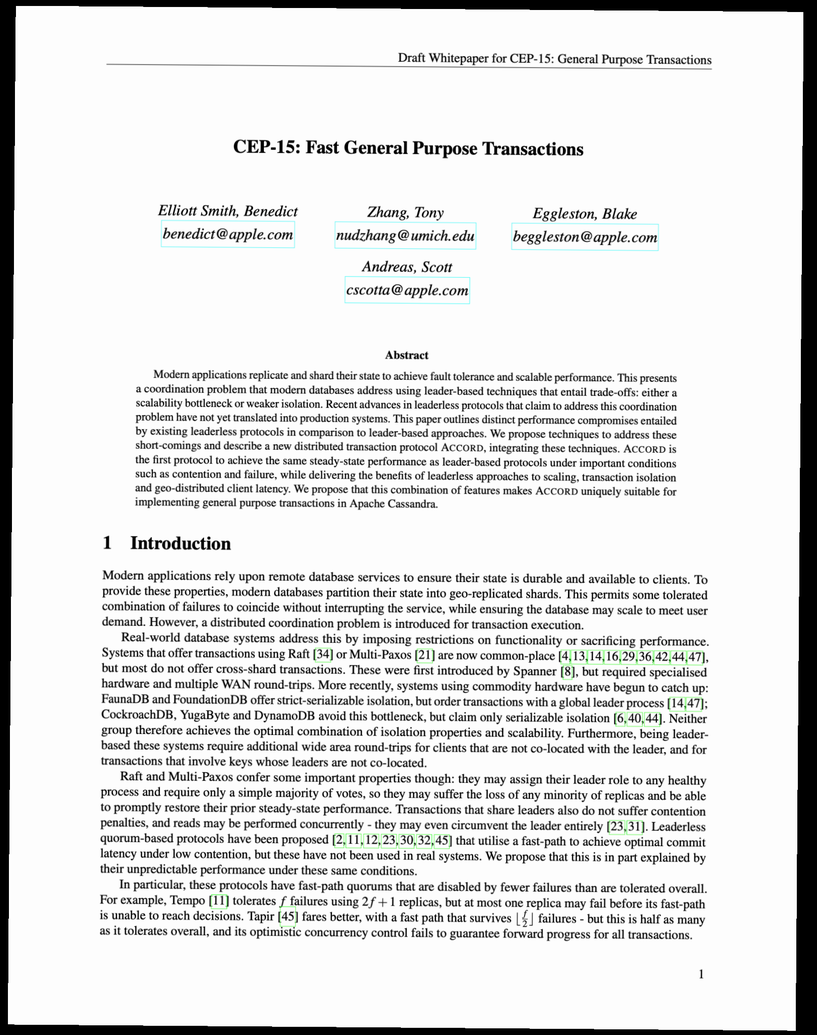
Transactional Database, Dynamo Foundations

- Transact over any subset of keys in the database, including across tables.
- Support strict serializability: strongest level of isolation possible.
- Optimal latency: one WAN round-trip for all transactions under normal operation.
- Optimal fault tolerance: Latency and performance resilient to a minority of replica failures.
- Scalability: No single point of coordination or bottleneck introduced.
- Portability: No specialized hardware required.

Accord

Consensus Between Parliaments

- **Leaderless Paxos:** Similarities to EPaxos, Tempo, Caesar.
- **Flexible Consensus Groups:** Variable per-transaction.
- **Hybrid Logical Time:** Ordered execution w/dependencies.
- **Single network round-trip:** Low-latency for multi-region apps.
- **Validation:** Specified with formal proof, validated via simulation.



Draft Whitepaper for CEP-15: General Purpose Transactions

CEP-15: Fast General Purpose Transactions

Elliott Smith, Benedict
benedict@apple.com

Zhang, Tony
nudzhang@umich.edu

Eggleston, Blake
beggleston@apple.com

Andreas, Scott
cscotta@apple.com

Abstract

Modern applications replicate and shard their state to achieve fault tolerance and scalable performance. This presents a coordination problem that modern databases address using leader-based techniques that entail trade-offs: either a scalability bottleneck or weaker isolation. Recent advances in leaderless protocols that claim to address this coordination problem have not yet translated into production systems. This paper outlines distinct performance compromises entailed by existing leaderless protocols in comparison to leader-based approaches. We propose techniques to address these short-comings and describe a new distributed transaction protocol ACCORD, integrating these techniques. ACCORD is the first protocol to achieve the same steady-state performance as leader-based protocols under important conditions such as contention and failure, while delivering the benefits of leaderless approaches to scaling, transaction isolation and geo-distributed client latency. We propose that this combination of features makes ACCORD uniquely suitable for implementing general purpose transactions in Apache Cassandra.

1 Introduction

Modern applications rely upon remote database services to ensure their state is durable and available to clients. To provide these properties, modern databases partition their state into geo-replicated shards. This permits some tolerated combination of failures to coincide without interrupting the service, while ensuring the database may scale to meet user demand. However, a distributed coordination problem is introduced for transaction execution.

Real-world database systems address this by imposing restrictions on functionality or sacrificing performance. Systems that offer transactions using Raft [34] or Multi-Paxos [21] are now common-place [4,13,14,16,29,36,42,44,47], but most do not offer cross-shard transactions. These were first introduced by Spanner [8], but required specialised hardware and multiple WAN round-trips. More recently, systems using commodity hardware have begun to catch up: FaunaDB and FoundationDB offer strict-serializable isolation, but order transactions with a global leader process [14,47]; CockroachDB, YugaByte and DynamoDB avoid this bottleneck, but claim only serializable isolation [6,40,44]. Neither group therefore achieves the optimal combination of isolation properties and scalability. Furthermore, being leader-based these systems require additional wide area round-trips for clients that are not co-located with the leader, and for transactions that involve keys whose leaders are not co-located.

Raft and Multi-Paxos confer some important properties though: they may assign their leader role to any healthy process and require only a simple majority of votes, so they may suffer the loss of any minority of replicas and be able to promptly restore their prior steady-state performance. Transactions that share leaders also do not suffer contention penalties, and reads may be performed concurrently - they may even circumvent the leader entirely [23,31]. Leaderless quorum-based protocols have been proposed [2,11,12,23,30,32,45] that utilise a fast-path to achieve optimal commit latency under low contention, but these have not been used in real systems. We propose that this is in part explained by their unpredictable performance under these same conditions.

In particular, these protocols have fast-path quorums that are disabled by fewer failures than are tolerated overall. For example, Tempo [11] tolerates f failures using $2f+1$ replicas, but at most one replica may fail before its fast-path is unable to reach decisions. Tapir [45] fares better, with a fast path that survives $\lfloor \frac{f}{2} \rfloor$ failures - but this is half as many as it tolerates overall, and its optimistic concurrency control fails to guarantee forward progress for all transactions.

Paper located at https://is.gd/cassandra_accord

Leaderless Transactions

Why Paxos?

- “No distinguished nodes” harmonious with Cassandra’s architecture
- Transact from any region with predictable latency (unlike Multi-Paxos)
- No scalability bottleneck on a distinguished transaction authority
- Leaderless design avoids complications of elections and failover
- Leaderless enables single round trip, optimal latency

Paper located at https://is.gd/cassandra_accord

Comparing Implementations

	Scale	Isolation	Multi-Cloud	Leaderless	Single Key Round-Trips				Multi Key Round-Trips			
					Local		Remote		Local		Remote	
					Read	Write	Read	Write	Read	Write	Read	Write
CockroachDB	Terabytes	Serializable	✓	✗	1	1	2	2	1	1	2-3	2-3
DynamoDB	Petabytes	Serializable	✗	✗	1	1	2	2	1	1	NA	NA
Spanner	Petabytes	Strict Serializable	✗	✗	0.5	1	0.5	2	0.5	1	0.5	2-3
Cassandra (2013)	Petabytes	Linearizable	✓	✓	2	4	2	4	NA	NA	NA	NA
Cassandra (2022)	Petabytes	Linearizable	✓	✓	1	2	1	2	NA	NA	NA	NA
Cassandra (2023)	Petabytes	Strict Serializable	✓	✓	1	1	1	1	1	1	1	1

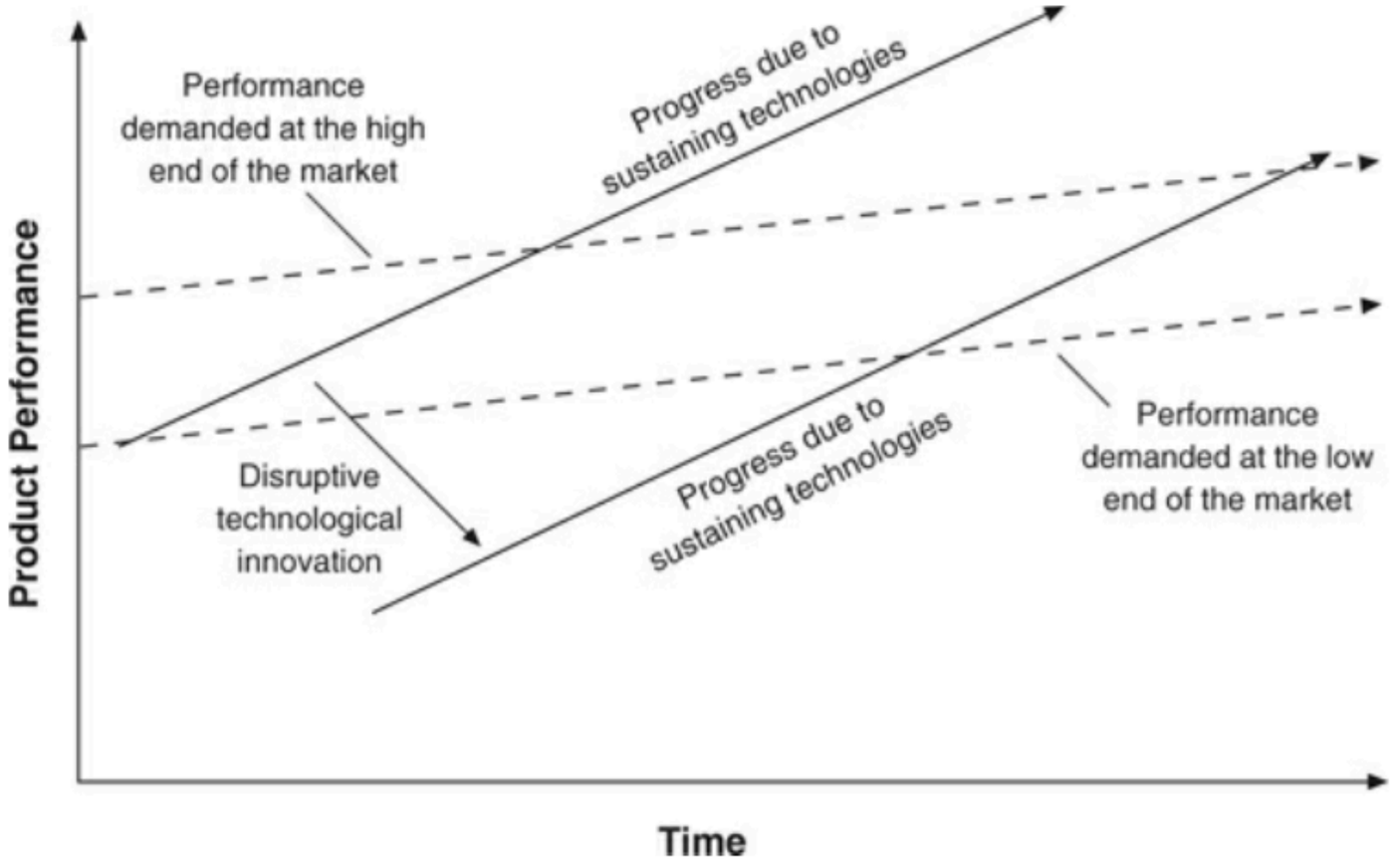
Paper located at https://is.gd/cassandra_accord

New Possibilities

Distributed Transactions Let Us…

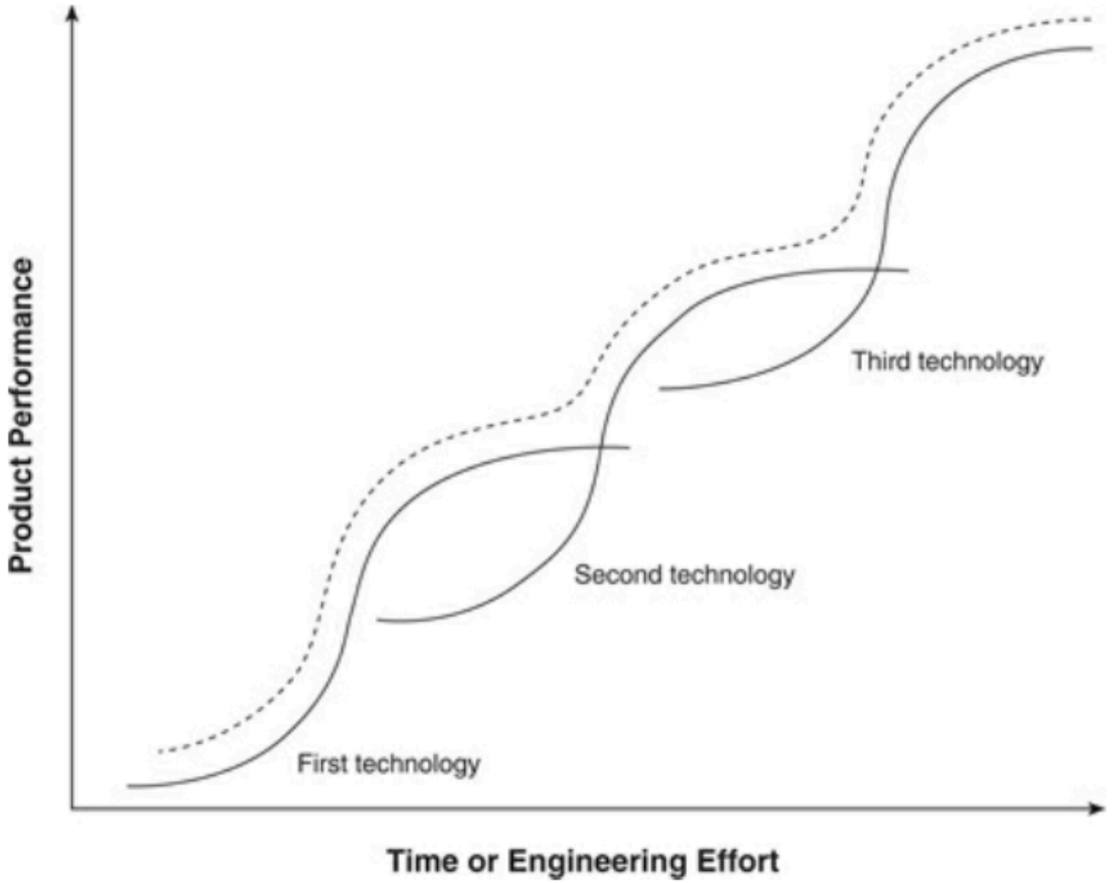
- **Rethink Indexes:** Transactional insert into base table and derived table.
- **Rethink Materialized Views:** Transactional insert into base, derived table.
- **Rethink Denormalization:** Enable enforcement of Foreign Key relationships.
- **Rethink MVCC:** Transactional versioning as a primitive to implement multi-version concurrency control, snapshot isolation.
- **Rethink Non-Relational:** FK relationships, snapshot isolation, ordered partitioning → cross-table joins.
- **Rethink a Dynamo-derived system as a transactional database.**

Figure I.1 The Impact of Sustaining and Disruptive Technological Change



The Innovator's Dilemma - Clayton M. Christensen. Harvard Business Review Press, 1997 - p. 16

Figure 2.5 The Conventional Technology S-Curve



The Innovator's Dilemma - Clayton M. Christensen. Harvard Business Review Press, 1997 - p. 78

Building on Dynamo's Legacy

Becoming a database that can…

- Serve petabytes of data and millions of queries/second at 99.9999% availability.
- Run active-active across up to 5 regions.
- Run in any datacenter or public cloud without specialized hardware.
- Execute leaderless transactions across the entirety of the database from any region.
- Be downloaded, learned from, and modified by anyone.

Dynamo at 15

What Worked and What's Next in Majority-Quorum Databases

C. Scott Andreas, Apple Inc.
HPTS 2022 – Pacific Grove, California