Data Systems for the Cloud

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

What is the cloud and what's different with it?

S3 & Dynamo: object stores

Aurora: transactional DBMS

BigQuery: analytical DBMS

Delta Lake: ACID over object stores

Outline

What is the cloud and what's different with it?

S3 & Dynamo: object stores

Aurora: transactional DBMS

BigQuery: analytical DBMS

Delta Lake: ACID over object stores

What is Cloud Computing?

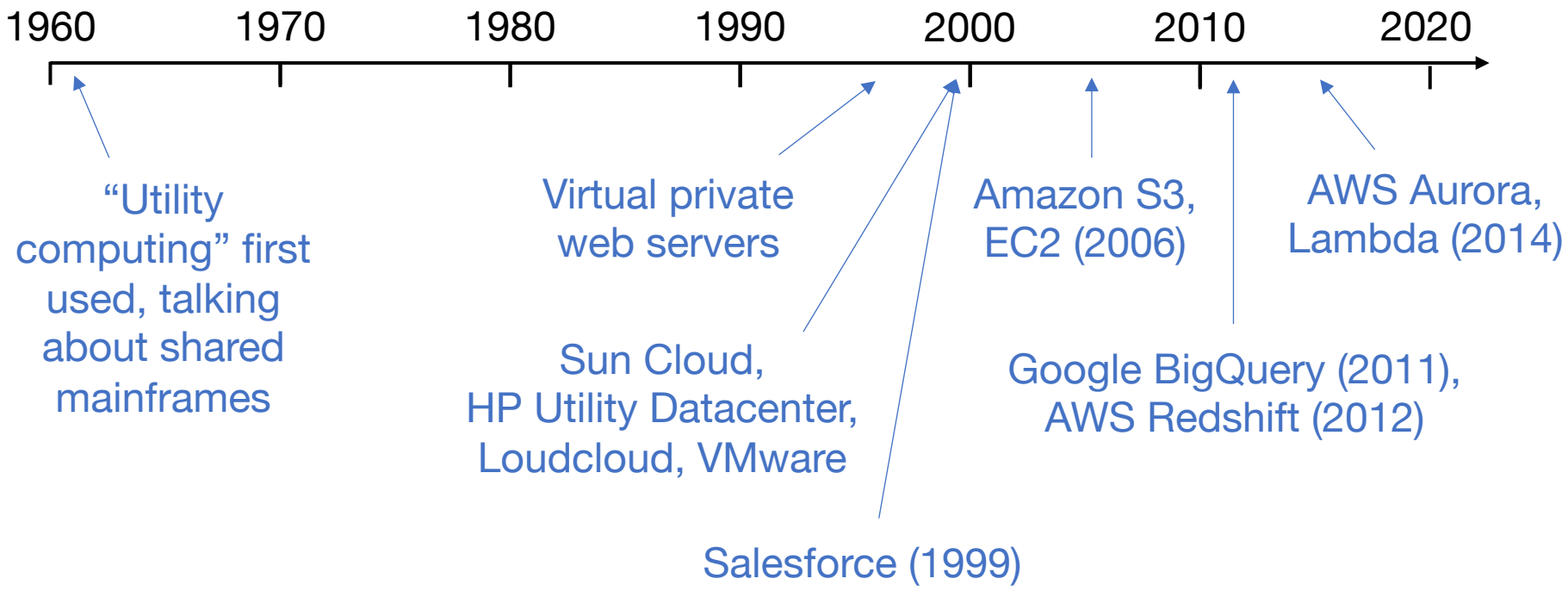
Computing as a service, managed by an external party

- **Software as a Service (SaaS):** application hosted by a provider, e.g. Salesforce, Gmail
- **Platform as a Service (PaaS):** APIs to program against, e.g. DB or web hosting
- **Infrastructure as a Service (IaaS):** raw computing resources, e.g. VMs on AWS

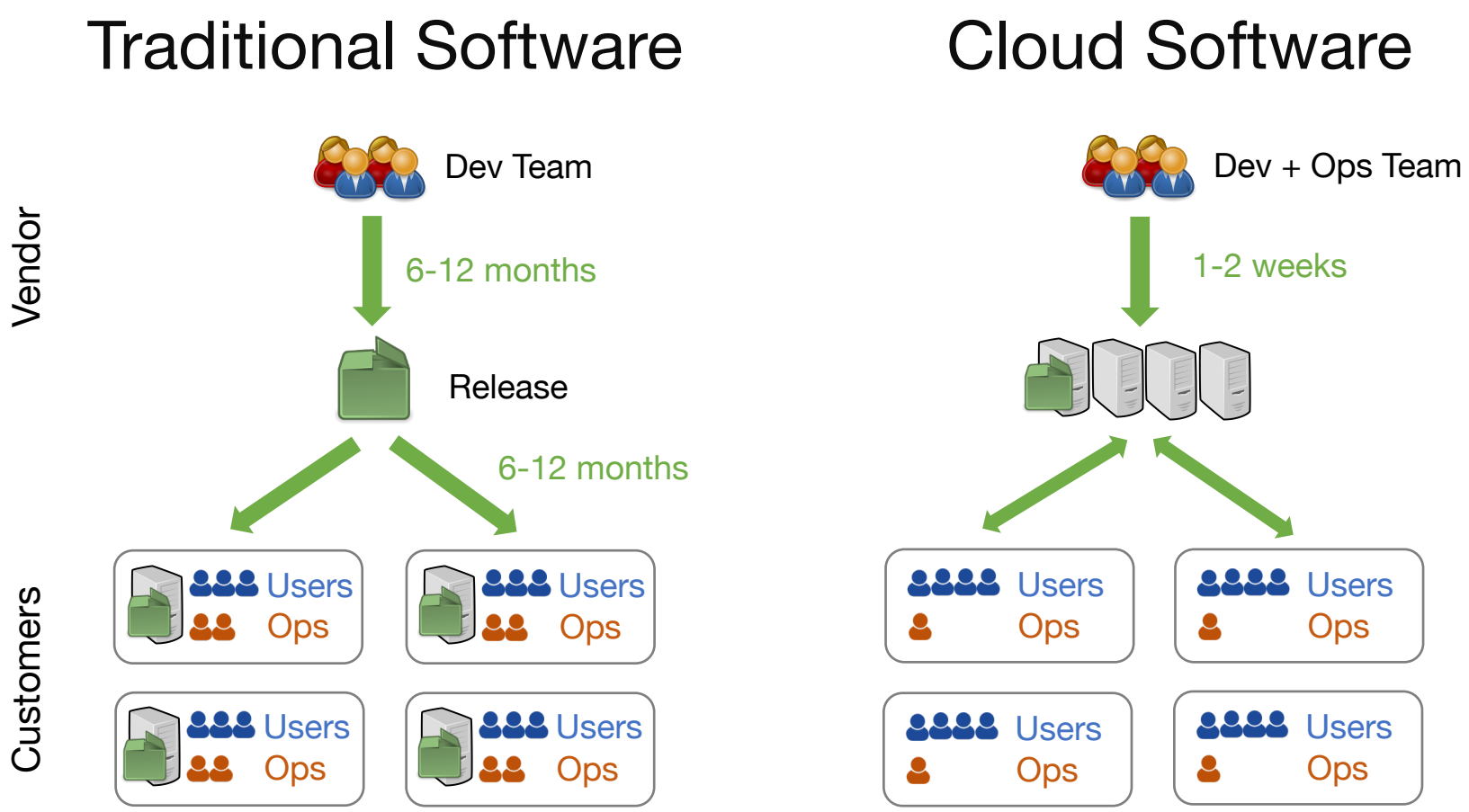
Large shift in industry over past 10-20 years!

History of Cloud Computing

Old idea, but became successful in the 2000s



Development Process



Why Might Customers Use Cloud Services?

Management built-in: more value than the software bits alone (security, availability, etc)

Elasticity: pay-as-you-go, scale on demand

Better features released faster

Differences in Building Cloud Software

+ Release cycle: send releases to users faster, get feedback faster

+ Only need to maintain 2 software versions (current & next), fewer configs than on-premise

– Upgrading without regressions: critical for users to trust service, as updates are forced

– Building a multitenant service: difficult scaling, security and performance isolation work

How Do These Factors Affect Data Systems?

Data systems already had to support many users robustly, but new challenges arise:

- **Much larger scale:** millions of users, VMs, …
- **Multitenancy:** users don't trust each other, so need stronger security, perf isolation, etc
- **Elasticity:** how can our system be elastic?
- **Updatability:** avoid regressions & downtime

Outline

What is the cloud and what's different with it?

S3 & Dynamo: object stores

Aurora: transactional DBMS

BigQuery: analytical DBMS

Delta Lake: ACID over object stores

S3, Dynamo & Object Stores

Goal: I just want to store some bytes reliably and cheaply for a long time period

Interface: key-value stores

» Objects have a key (e.g. `bucket/imgs/1.jpg`) and value (arbitrary bytes)

» Values can be up to a few TB in size

» Can only do operations on 1 key atomically

Consistency: eventual consistency

Store trillions of objects and exabytes of data

Example: S3 API

PUT(key, value): write object with a key
  » Atomic update: replaces the whole object

GET(key, [range]): return object with a key
  » Can optionally read a byte range in the object

LIST([startKey]): list keys in a bucket in lexicographic order, starting at a given key
  » Limit of 1000 returned keys per call

S3 Consistency Model

Eventual consistency: different readers may see different versions of the same object

Read-your-own-writes for new PUT: if you GET a *new* object that you PUT, you see it

» Unless you had previously called GET while it was missing, in which case you might not!

Why These Choices?

The primary goal is scale: keep the interface very simple to support trillions of objects

» No cross-object operations except LIST!

Mostly target immutable or rarely changing data, so consistency is not as important

Can try to build stronger consistency on top

Implementing Object Stores

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

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

Goals

Query Model: simple read and write operations to a data item that is uniquely identified by a key. State is stored as binary objects (i.e., blobs) identified by unique keys. No operations span multiple data items and there is no need for relational schema. This requirement is based on the observation that a significant portion of Amazon's services can work with this simple query model and do not need any relational schema.

Goals

Efficiency: The system needs to function on a commodity hardware infrastructure. In Amazon's platform, services have stringent latency requirements which are in general measured at the 99.9^{th} percentile of the distribution. Given that state access plays a crucial role in service operation the storage system must be capable of meeting such stringent SLAs (see Section 2.2 below). Services must be able to configure Dynamo such that they consistently achieve their latency and throughput requirements. The tradeoffs are in performance, cost efficiency, availability, and durability guarantees.

Goals

Other Assumptions: Dynamo is used only by Amazon's internal services. Its operation environment is assumed to be non-hostile and there are no security related requirements such as authentication and authorization. Moreover, since each service uses its distinct instance of Dynamo, its initial design targets a scale of up to hundreds of storage hosts.

Obviously different for S3!

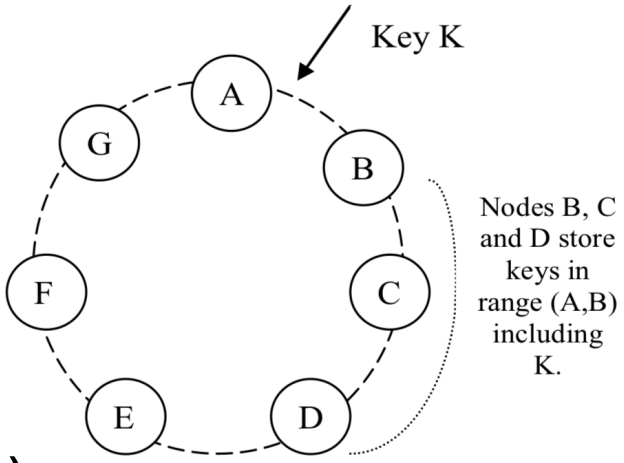
Dynamo Implementation

Commodity nodes with local storage on disks

Nodes form a “ring” to split up the key space among them

» Actually, each node covers many ranges (over-partitioning)

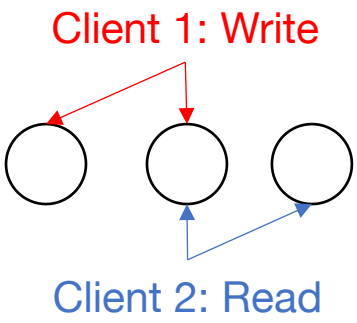
Use quorums and gossip to manage updates to each key



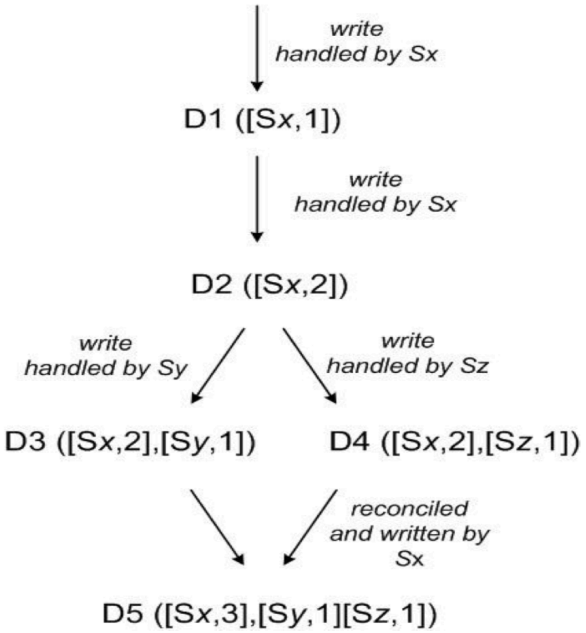
Reads and Writes to Dynamo

Quorums with configurable # of writers and readers required for success

- » E.g. 3 nodes, write to 2, read from 2
- » E.g. 3 nodes, write to 2, read from 1 (weaker consistency!)



Nodes gossip & merge updates in an application-specific way

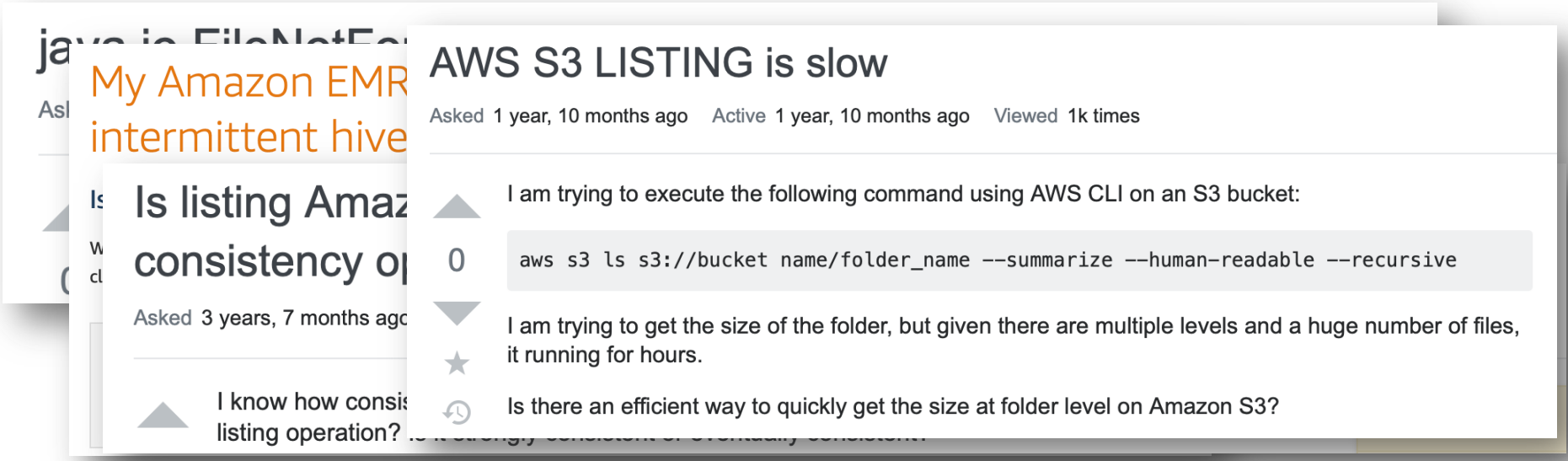


Usage of Object Stores

Very widely used (probably the largest storage systems in the world)

But the semantics can be complex

» E.g. many users try to mount these as file systems but they're not the same



Outline

What is the cloud and what's different with it?

S3 & Dynamo: object storage

Aurora: transactional DBMS

BigQuery: analytical DBMS

Delta Lake: ACID over object stores

Amazon Aurora

Goal: I want a transactional DBMS managed by the cloud vendor

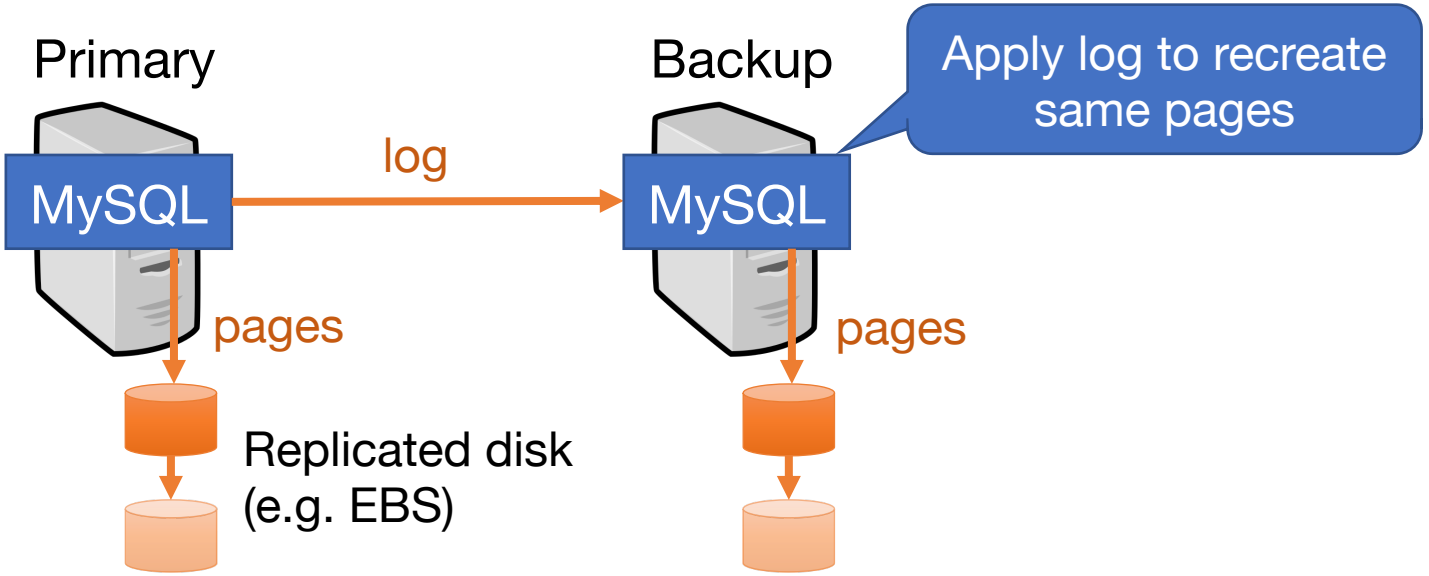
Interface: same as MySQL/Postgres
» ODBC, JDBC, etc

Consistency: strong consistency (similar to traditional DBMSes)

Some of the largest & most profitable cloud services

Initial Attempt at DBMS on AWS

Just run an existing DBMS (e.g. MySQL) on cloud VMs, and use replicated disk storage



Same thing users would do on-premise

Problems with This Model

Elasticity: doesn't leverage the elastic nature of the cloud, or give users elasticity

Efficiency: mirroring and disk-level replication is expensive at global scale

Inefficiency of Mirrored DBMS

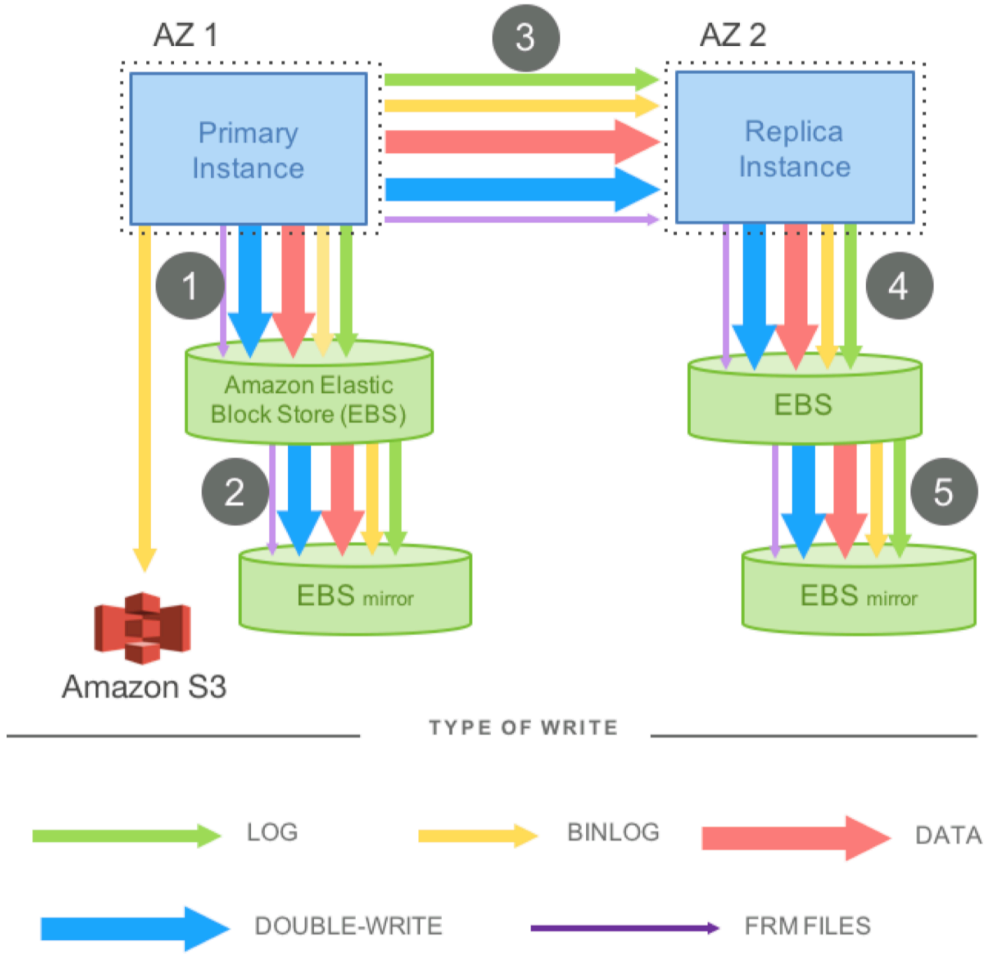


Figure 2: Network IO in mirrored MySQL

Write amplification: each write at app level results in many writes to physical storage

For Aurora, Amazon wanted “4 out of 6” quorums (3 zones and 2 nodes in each zone)

Aurora's Design

Implement replication at a higher level: only replicate the redo log (not disk blocks)

Enable elastic frontend and backend by decoupling API & storage servers

» Lower cost & higher performance per tenant

Aurora's Design

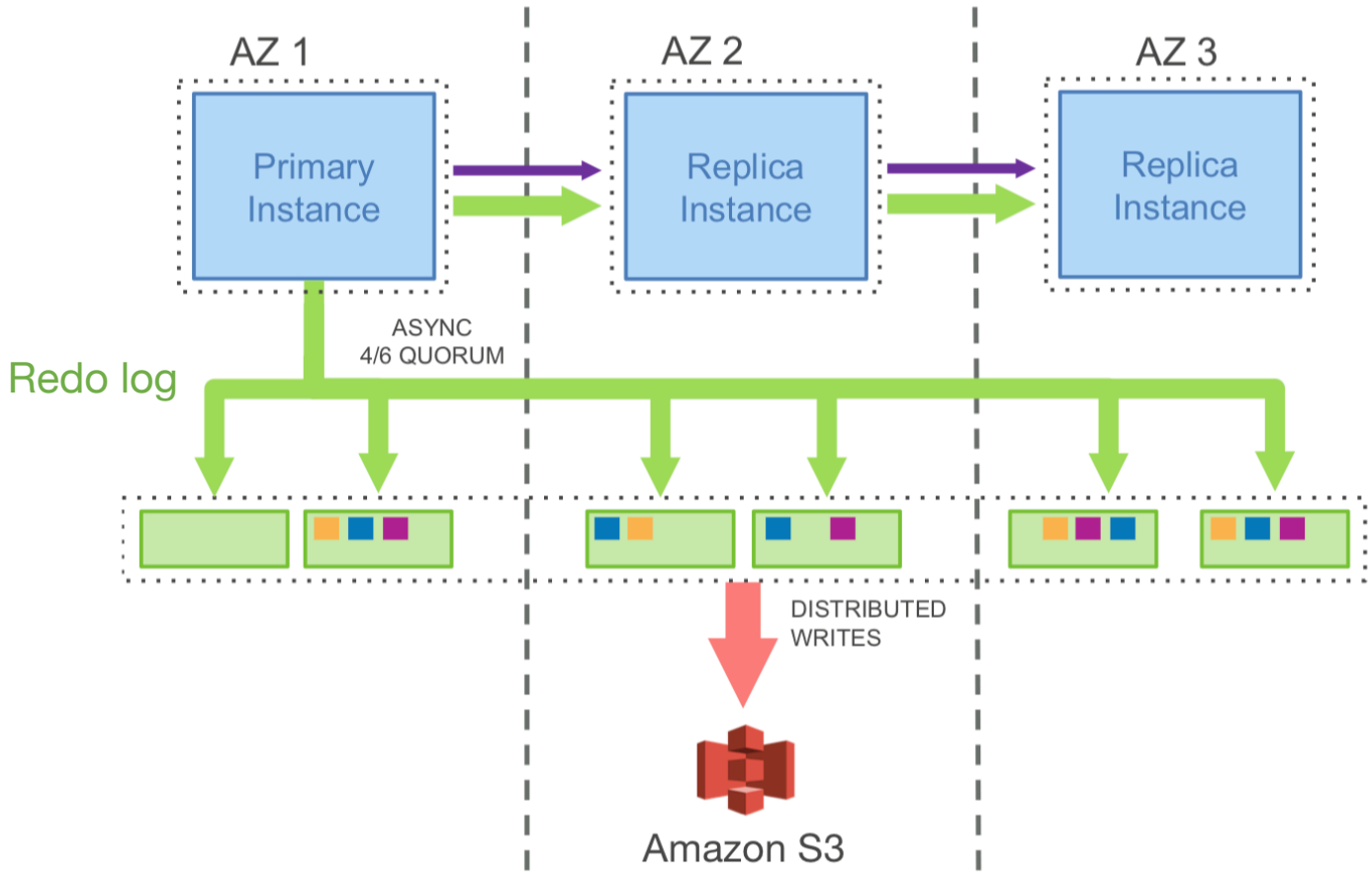


Figure 3: Network IO in Amazon Aurora

Design Details

Logging uses async quorum: wait until 4 of 6 nodes reply (faster than waiting for all 6)

Each storage node takes the log and rebuilds the DB pages locally

Care taken to handle incomplete logs due to async quorums

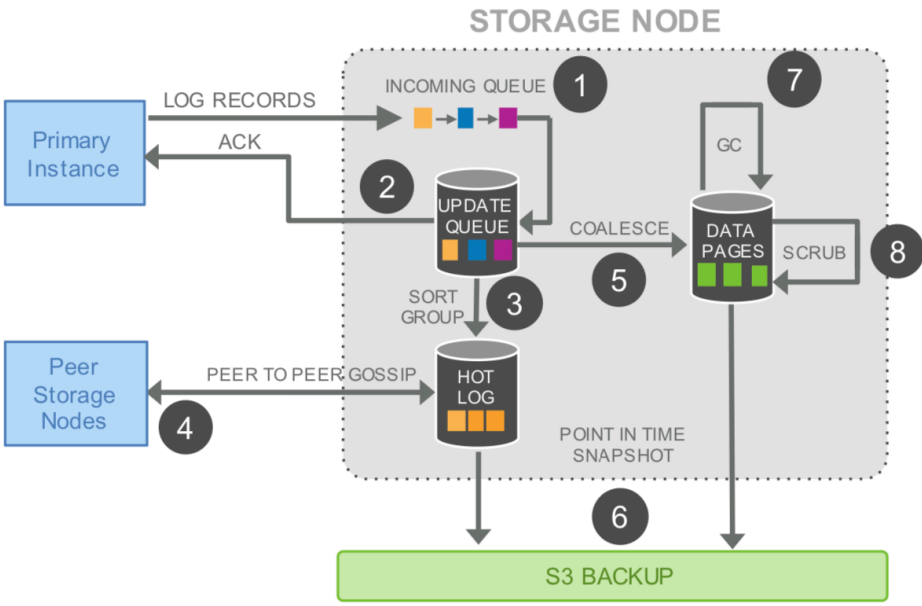
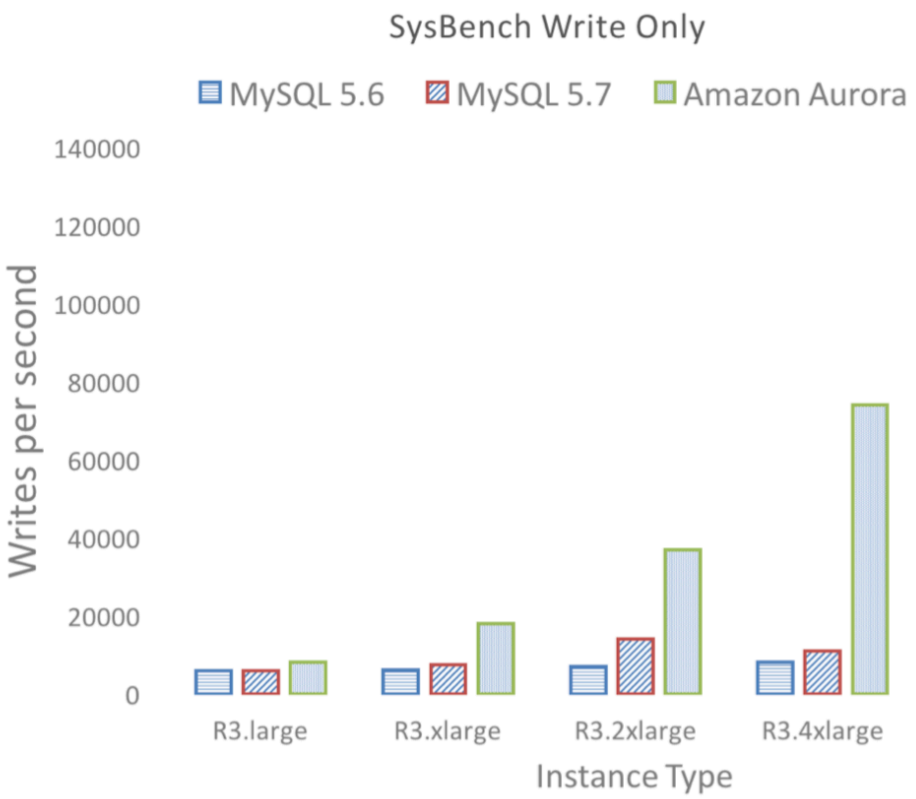
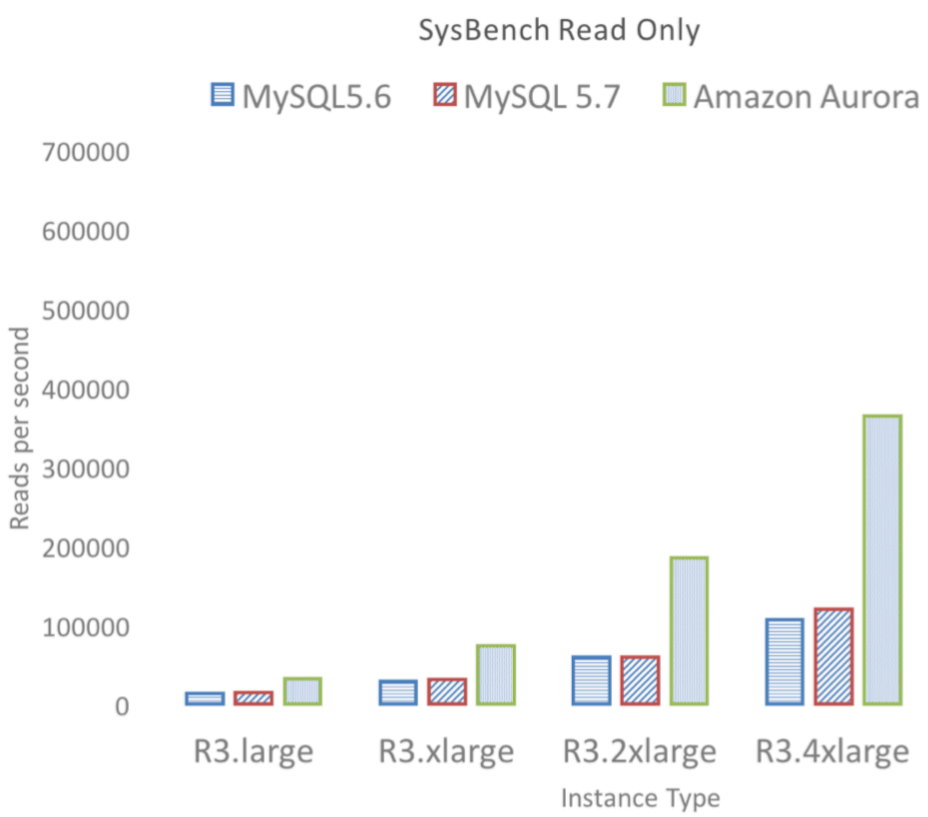


Figure 4: IO Traffic in Aurora Storage Nodes

Performance





Other Features from this Design

Rapidly add or remove read replicas

Efficient DB recovery, cloning and rollback (use a prefix of the log and older pages)

Serverless Aurora (only pay when actively running queries)

Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, Xiaofeng Bao

Amazon Web Services

ABSTRACT

Amazon Aurora is a relational database service for OLTP workloads offered as part of Amazon Web Services (AWS). In this paper, we describe the architecture of Aurora and the design considerations leading to that architecture. We believe the central constraint in high throughput data processing has moved from compute and storage to the network. Aurora brings a novel architecture to the relational database to address this constraint, most notably by pushing redo processing to a multi-tenant scale-out storage service, purpose-built for Aurora. We describe how doing so not only reduces network traffic, but also allows for fast crash recovery, failovers to replicas without loss of data, and fault-tolerant, self-healing storage. We then describe how Aurora achieves consensus on durable state across numerous storage nodes using an efficient asynchronous scheme, avoiding expensive and chatty recovery protocols. Finally, having operated Aurora as a production service for over 18 months, we share lessons we have learned from our customers on what modern cloud applications expect from their database tier.

Keywords

Databases; Distributed Systems; Log Processing; Quorum Models; Replication; Recovery; Performance; OLTP

1. INTRODUCTION

IT workloads are increasingly moving to public cloud providers. Significant reasons for this industry-wide transition include the

The I/O bottleneck faced by traditional database systems changes in this environment. Since I/Os can be spread across many nodes and many disks in a multi-tenant fleet, the individual disks and nodes are no longer hot. Instead, the bottleneck moves to the network between the database tier requesting I/Os and the storage tier that performs these I/Os. Beyond the basic bottlenecks of packets per second (PPS) and bandwidth, there is amplification of traffic since a performant database will issue writes out to the storage fleet in parallel. The performance of the outlier storage node, disk or network path can dominate response time.

Although most operations in a database can overlap with each other, there are several situations that require synchronous operations. These result in stalls and context switches. One such situation is a disk read due to a miss in the database buffer cache. A reading thread cannot continue until its read completes. A cache miss may also incur the extra penalty of evicting and flushing a dirty cache page to accommodate the new page. Background processing such as checkpointing and dirty page writing can reduce the occurrence of this penalty, but can also cause stalls, context switches and resource contention.

Transaction commits are another source of interference; a stall in committing one transaction can inhibit others from progressing. Handling commits with multi-phase synchronization protocols such as 2-phase commit (2PC) [3][4][5] is challenging in a cloud-scale distributed system. These protocols are intolerant of failure and high-scale distributed systems have a continual "background

Outline

What is the cloud and what's different with it?

S3 & Dynamo: object stores

Aurora: transactional DBMS

BigQuery: analytical DBMS

Delta Lake: ACID over object stores

Google BigQuery

Goal: I want a cheap & fast analytical DBMS managed by the cloud vendor

Interface: SQL, JDBC, ODBC, etc

Consistency: depends on storage chosen (object stores or richer table storage)

Traditional Data Warehouses

Provision a fixed set of nodes that have both storage and computing

- » Big servers with lots of disks, etc
- » Makes sense when buying servers on-premise

Problem: no elasticity!

Interestingly, this was the model chosen by AWS Redshift initially (using ParAccel)

BigQuery and Other Elastic Analytics Systems

Separate compute and storage

- » One set of nodes (or the cloud object store) stores data, usually over 1000s of nodes
- » Separate set of nodes handle queries (again, possibly scaling out to 1000s)

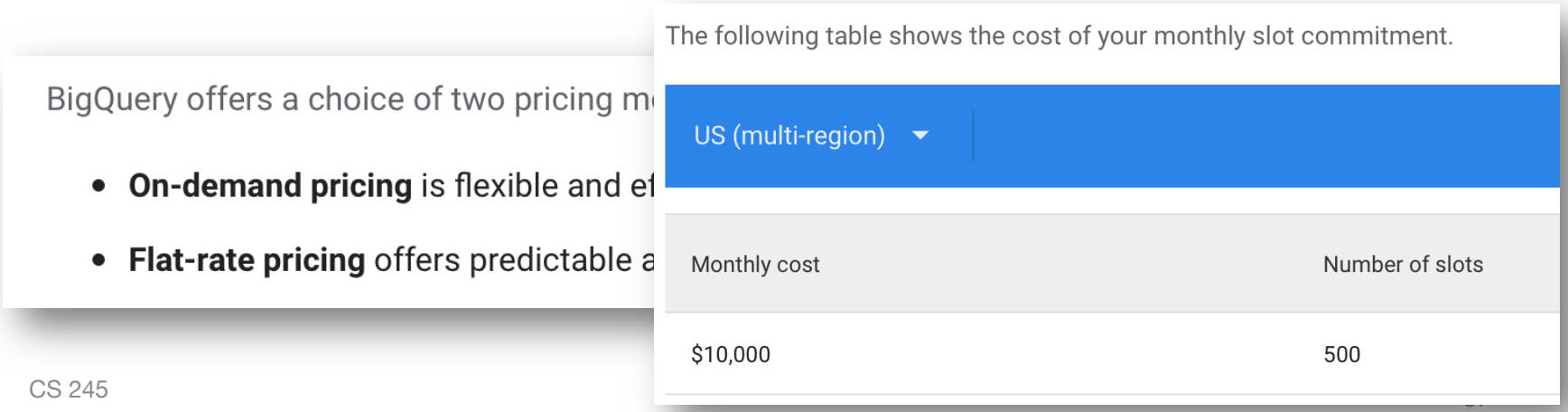
Users pay separately for storage & queries

Get performance of 1000s of servers to run a query, but only pay for a few seconds of use

Results

These elastic services generally provide better performance and cost for ad-hoc small queries than launching a cluster

For big organizations or long queries, paying per query can be challenging, so these services let you bound total # of nodes



Outline

What is the cloud and what's different with it?

S3 & Dynamo: object stores

Aurora: transactional DBMS

BigQuery: analytical DBMS

Delta Lake: ACID over object stores

Delta Lake Motivation

Object stores are the largest & lowest-cost storage systems, but their semantics make it hard to manage mutable datasets

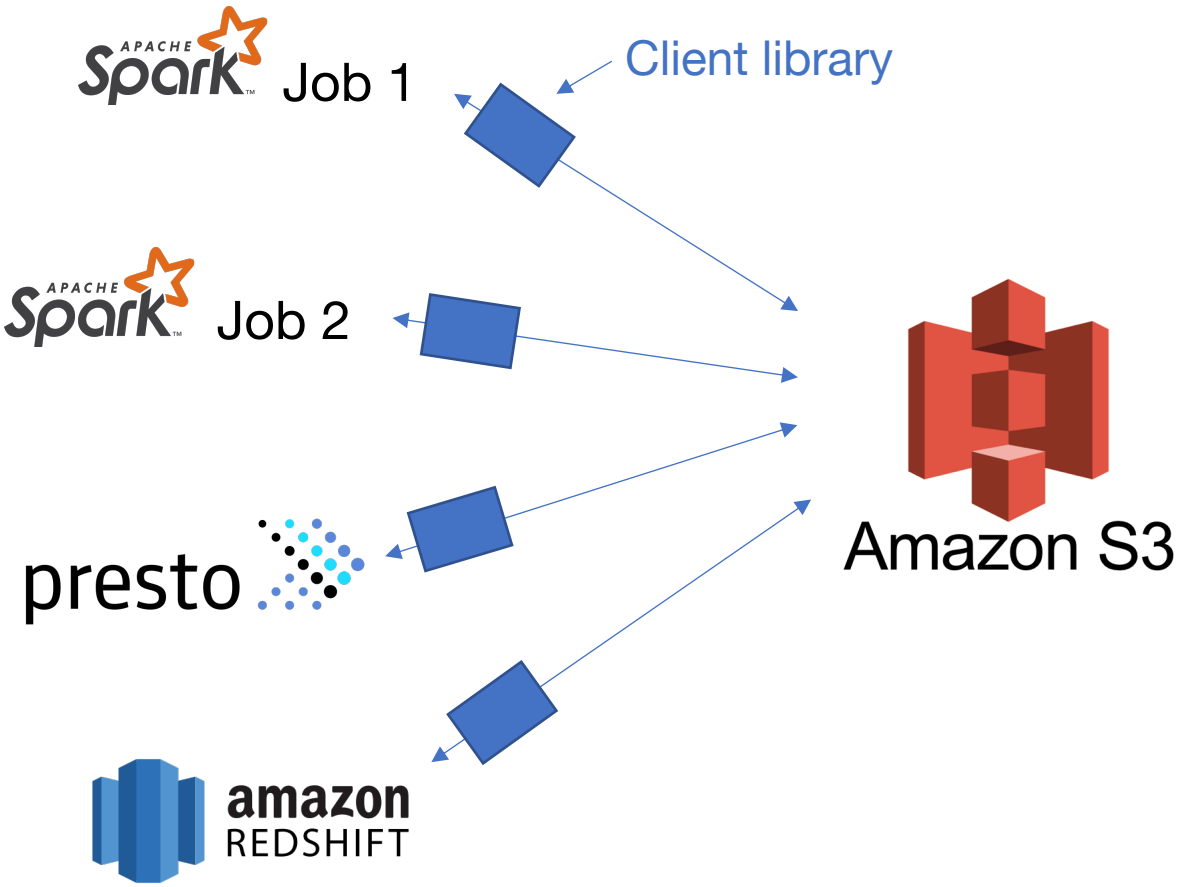
Goal: analytical table storage over object stores, built as a client library (no other services)

Interface: relational tables with SQL queries

Consistency: serializable ACID transactions

Open source at https://delta.io

Setup



Naïve Way to Use Object Stores for Tables

"Just a bunch of objects": a table is a set of files (maybe partitioned on some fields)

```
mytable/date=2020-01-01/p1.parquet
                       /p2.parquet
       /date=2020-01-02/p1.parquet
                       /p2.parquet
                       /p3.parquet
       /date=2020-01-03/p1.parquet
...
```

Columnar files of records with date=2020-01-01

Columnar files of records with date=2020-01-02

…

Problems with “Just Objects”

No multi-object transactions

- » Hard to insert multiple objects at once (what if your load job crashes partway through?)
- » Hard to update multiple objects at once (e.g. delete a user or fix their records)
- » Hard to change data layout & partitioning

Poor performance

- » LIST is expensive (only 1000 results/request!)
- » Can’t do streaming inserts (too many small files)
- » Expensive to load metadata (e.g. column stats)

Example Problems

Keep getting FileNotFound for tempView

[redacted]

CRITICAL production problem: inconsistent job e...

[redacted]

Appending new data to a partitioned table

[redacted]

Different field types cause conflicting schemas w...

[redacted]

Example Problems

Extremely slow dataframe loading

Commands Blocked on Metadata Operations

Concatenate small files

how to control number of parquet files within par...

Delta Lake's Approach

Can we implement a transaction log on top of the object store to retain its scale & reliability but provide stronger semantics?

Inspiration: Bolt-On Consistency

Bolt-on Causal Consistency

Peter Bailis[†], Ali Ghodsi[†,‡], Joseph M. Hellerstein[†], Ion Stoica[†]
[†] UC Berkeley [‡] KTH/Royal Institute of Technology

Shallow men believe in luck... Strong men believe in cause and effect.—Ralph Waldo Emerson

ABSTRACT

We consider the problem of separating consistency-related safety properties from availability and durability in distributed data stores via the application of a "bolt-on" shim layer that upgrades the safety of an underlying general-purpose data store. This shim provides the same consistency guarantees atop a wide range of widely deployed but often inflexible stores. As causal consistency is one of the strongest consistency models that remain available during system partitions, we develop a shim layer that upgrades eventually consistent stores to provide convergent causal consistency. Accordingly, we leverage widely deployed eventually consistent infrastructure as a common substrate for providing causal guarantees. We describe algorithms and shim implementations that are suitable for a large class of application-level causality relationships and evaluate our techniques using an existing, production-ready data store and with real-world explicit causality relationships.

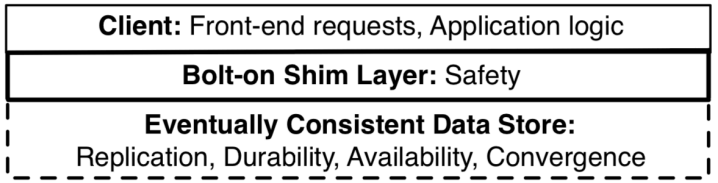


Figure 1: Separation of concerns in the bolt-on architecture. In this work, a narrow shim layer upgrades an underlying eventually consistent data store to provide causal consistency.

will eventually happen (replicas agree) [5]. It does not provide *safety* guarantees: it is impossible for a system to violate eventual consistency at any discrete moment because the system may converge in the future [7]. By itself, eventual consistency is a weak consistency model. If users want stronger guarantees, stores may provide stronger models that, in addition to convergence, provide safety. However, exact guarantees differ across stores and sometimes between releases of the same stores.

In this paper we adopt a general approach that separates architectural concerns of liveness, replication, and durability from the

Delta Lake Implementation

Table = directory of *data objects*, with a set of *log objects* stored in `_delta_log` subdir

» Log specifies **which** data objects are part of the table at a given version

One log object for each write transaction, in order: `000001.json`, `000002.json`, etc

Periodic *checkpoints* of the log in Parquet format contain object list + column statistics

Delta Table Example

```
mytable/date=2020-01-01/1b8a32d2ad.parquet
                       /a2dc5244f7.parquet
                       /f52312dfae.parquet
                       /ba68f6bd4f.parquet
       /_delta_log/00001.json
                  /00002.json
                  /00003.json
                  /00003.parquet
                  /00004.json
                  /00005.json
                  /_last_checkpoint
```

Data objects (partitioned by `date` field)

Log records and checkpoints

Contains `{version: "00003"}`

Transaction's operations, e.g.,

```
add date=2020-01-01/a2dc5244f7f7.parquet
add date=2020-01-02/ba68f6bd4f1e.parquet
```

Coalesces log records 1–3

Log Record Types

Add data object + its column statistics

Remove data object

Change metadata, e.g. table schema or Delta Lake format version

A few others for streaming writes (allows treating a table like a message bus)

Writing to Delta Lake

1) Add new objects in the data directories; readers will ignore them because the log has no add entries for them

2) Try to add a new log record with the next valid log record number (e.g. `00006.json`)
   - » Various ways to make this atomic per cloud

3) Optional: write a new Parquet checkpoint

What if one of these steps fails?

What Kind of Concurrency Approach is This?

Optimistic! Even simpler than validation

Also MVCC: keep old data versions around

Why is this okay for Delta Lake's workloads?

Reading from Delta Lake

1) Read the `_last_checkpoint` object to find a checkpoint number

2) Read that Parquet file, and use LIST to find any newer `.json` log records after it

3) Determine which objects are “add”ed but not “remove”d from those logs and read those
   - » Use column min/max stats to prune data

What if one step sees old versions of that data?

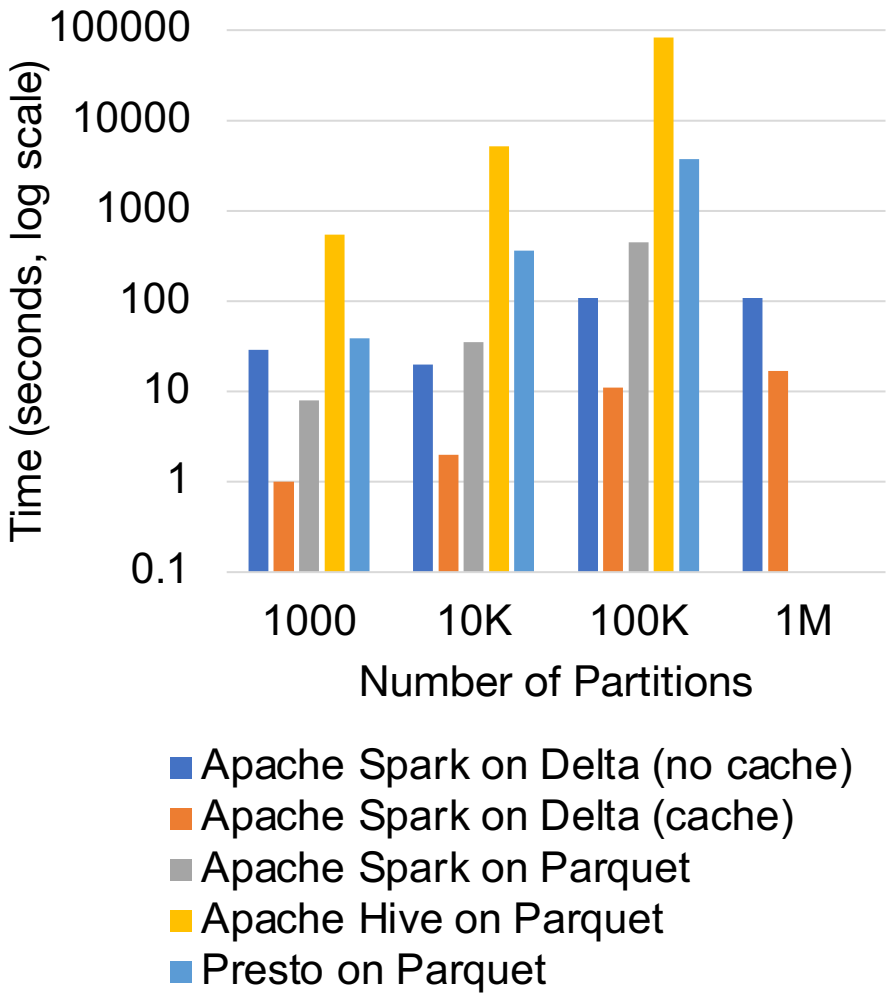
Isolation Levels

Transactions with writes are serializable: one serial order, given by log record numbers

Read transactions can get either snapshot isolation (read older version) or serializability (by adding a dummy write)

Takeaway: by using atomic operations on just one object at a time (last log record key), we got ACID transactions for a whole table!

Impact on Performance



Reading the list of object names from a Parquet file much faster than making many LIST operations

Reading column stats from this file is also faster than range GETs on each object

Other Features from this Design

Caching data & log objects on workers is safe because they are immutable

Time travel: can query or restore an old version of the table while those objects are retained

Background optimization: compact small writes or change data ordering (e.g. Z-order) without affecting concurrent readers

Audit logging: who wrote to the table?

Applications & Impact

Delta Lake now manages exabytes of data (>60% of Databricks' workload in 3 years)!

Reduced support escalations relating to cloud storage from ~50% to nearly none

Largest single tables hold exabytes of data across billions of data objects

Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores

Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Świtakowski, Michał Szafrański, Xiao Li, Takuya Ueshin, Mostafa Mokhtar, Peter Boncz[1], Ali Ghodsi[2], Sameer Paranjpye, Pieter Senster, Reynold Xin, Matei Zaharia[3]
Databricks, [1]CWI, [2]UC Berkeley, [3]Stanford University
delta-paper-authors@databricks.com

ABSTRACT

Cloud object stores such as Amazon S3 are some of the largest and most cost-effective storage systems on the planet, making them an attractive target to store large data warehouses and data lakes. Unfortunately, their implementation as key-value stores makes it difficult to achieve ACID transactions and high performance: metadata operations such as listing objects are expensive, and consistency guarantees are limited. In this paper, we present Delta Lake, an open source ACID table storage layer over cloud object stores initially developed at Databricks. Delta Lake uses a transaction log that is compacted into Apache Parquet format to provide ACID properties, time travel, and significantly faster metadata operations for large tabular datasets (e.g., the ability to quickly search billions of table partitions for those relevant to a query). It also leverages this design to provide high-level features such as automatic data layout optimization, upserts, caching, and audit logs. Delta Lake tables can be accessed from Apache Spark, Hive, Presto, Redshift and other systems. Delta Lake is deployed at thousands of Databricks customers that process exabytes of data per day, with the largest instances managing exabyte-scale datasets and billions of objects.

The major open source "big data" systems, including Apache Spark, Hive and Presto [45, 52, 42], support reading and writing to cloud object stores using file formats such as Apache Parquet and ORC [13, 12]. Commercial services including AWS Athena, Google BigQuery and Redshift Spectrum [1, 29, 39] can also query directly against these systems and these open file formats.

Unfortunately, although many systems support reading and writing to cloud object stores, achieving *performant* and *mutable* table storage over these systems is challenging, making it difficult to implement data warehousing capabilities over them. Unlike distributed filesystems such as HDFS [5], or custom storage engines in a DBMS, most cloud object stores are merely key-value stores, with no cross-key consistency guarantees. Their performance characteristics also differ greatly from distributed filesystems and require special care.

The most common way to store relational datasets in cloud object stores is using columnar file formats such as Parquet and ORC, where each table is stored as a set of objects (Parquet or ORC "files"), possibly clustered into "partitions" by some fields (e.g., a separate set of objects for each date) [45]. This approach can offer acceptable performance for scan workloads as long as the object files are moderately large. However, it creates both *correctness* and

Other “Bolt-On” Systems

Apache Hudi (at Uber) and Iceberg (at Netflix) also offer table storage on S3

Google BigTable was built over GFS

Filesystems that use S3 as a block store (e.g. early Hadoop s3:/, Goofys, MooseFS)

Conclusion

Cloud computing requires changes in data management systems

- » Elasticity with separate compute & storage
- » Very large scale
- » Multitenancy: security, performance isolation
- » Updating without regressions

Can design and analyze these systems using the ideas we saw!