COMP5349 – Cloud Computing

Week 10: Cloud Storage and Database Service

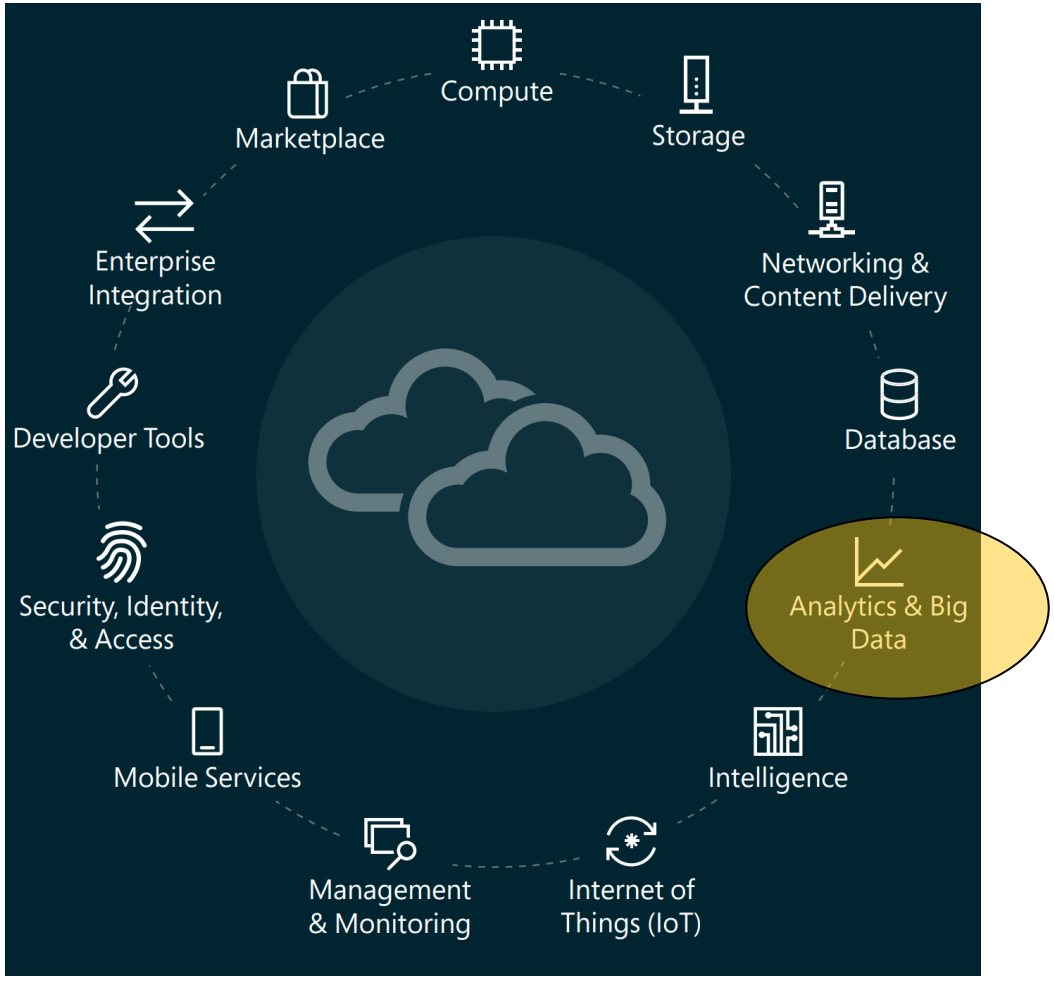
Dr. Ying Zhou
School of Computer Science

Outline

- **Cloud Storage and Database Services**
- Google Bigtable
- Windows Azure Storage
- Amazon Dynamo
- Amazon Aurora

Some slides are based on COMP5338, 2018 (Y. Zhou)

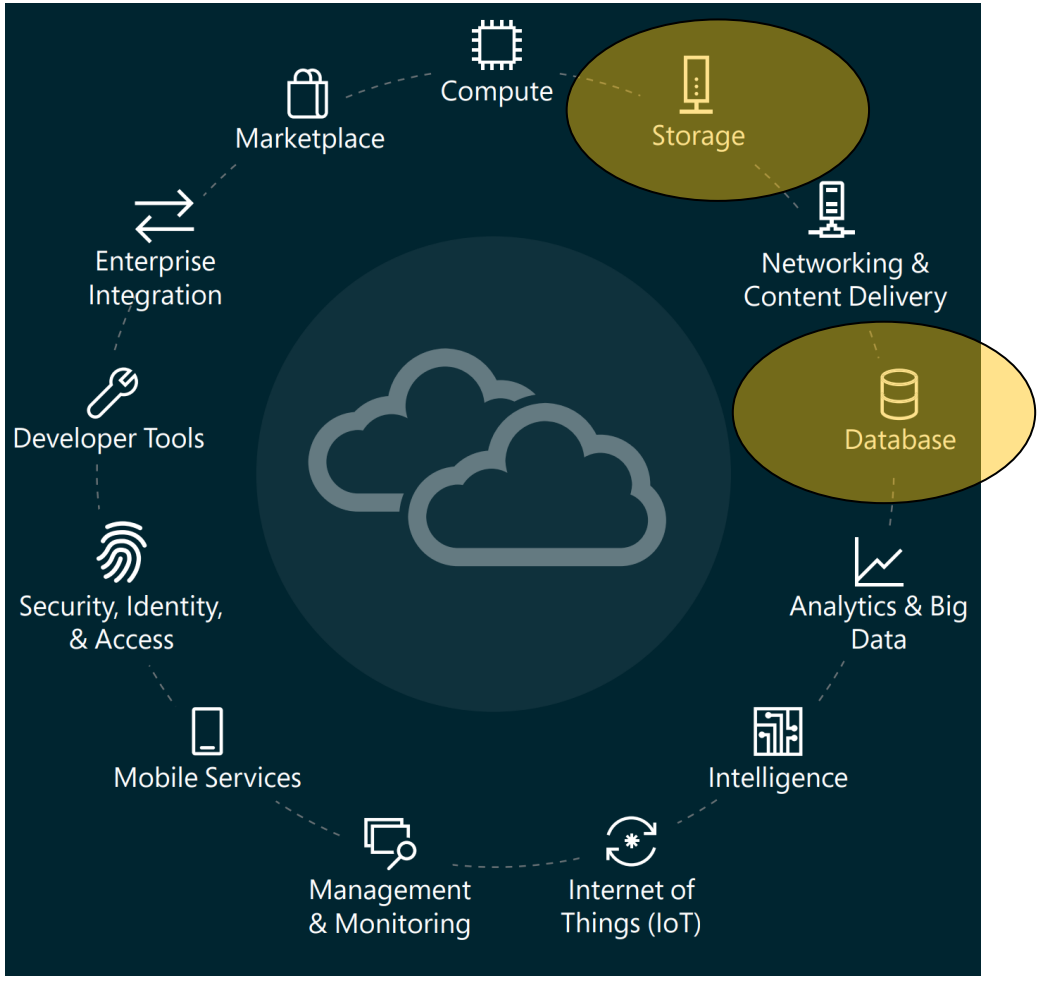
Cloud Analytics Services



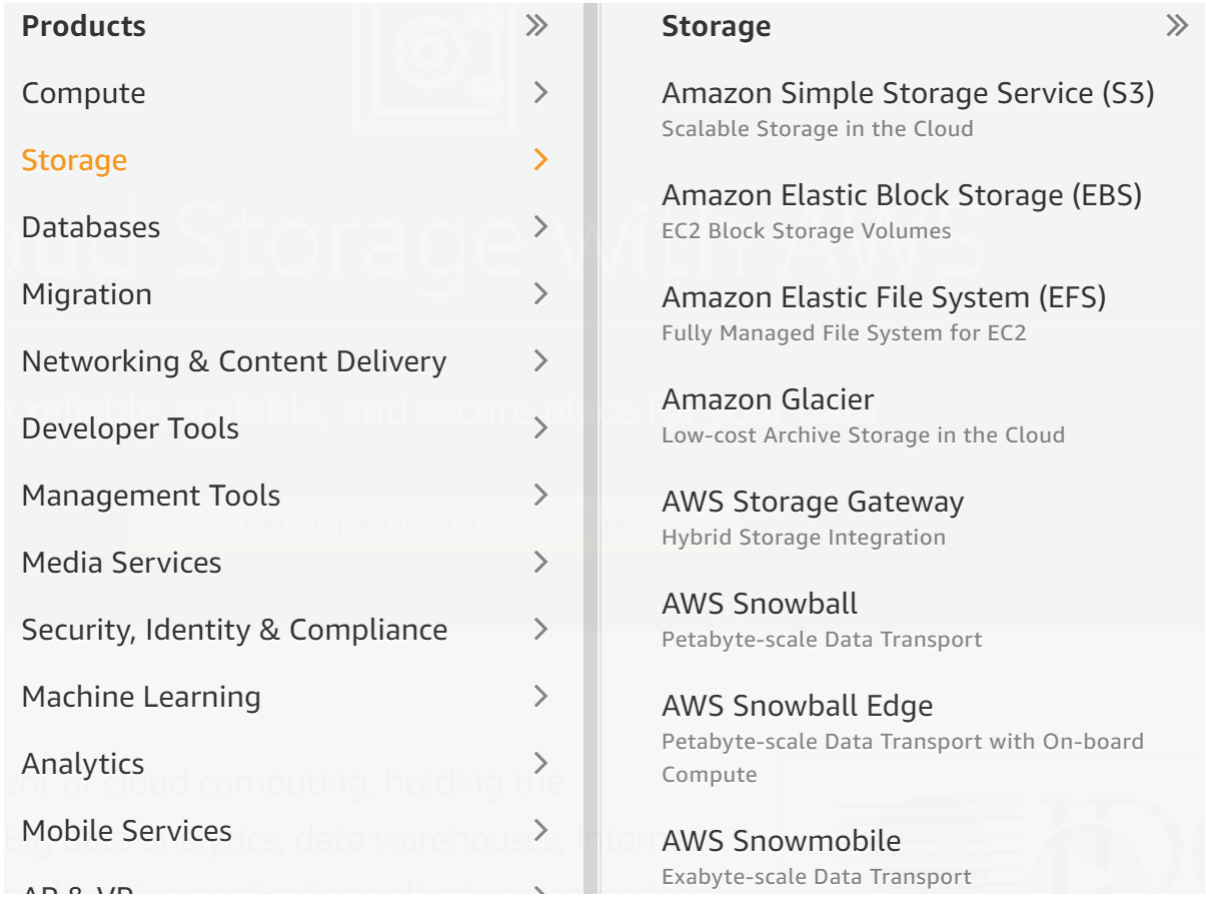
Cloud Analytic Services

IF YOU WANT...	USE THIS
A fully managed, elastic data warehouse with security at every level of scale at no extra cost	SQL Data Warehouse
A fully managed, fast, easy and collaborative Apache® Spark™ based analytics platform optimized for Azure	Azure Databricks
A fully managed cloud Hadoop and Spark service backed by 99.9% SLA for your enterprise	HDInsight
A data integration service to orchestrate and automate data movement and transformation	Data Factory
Open and elastic AI development spanning the cloud and the edge	Machine Learning
Real-time data stream processing from millions of IoT devices	Stream Analytics
A fully managed on-demand pay-per-job analytics service with enterprise-grade security, auditing, and support	Data Lake Analytics
Enterprise grade analytics engine as a service	Azure Analysis Services
A hyper-scale telemetry ingestion service that collects, transforms, and stores millions of events	Event Hubs

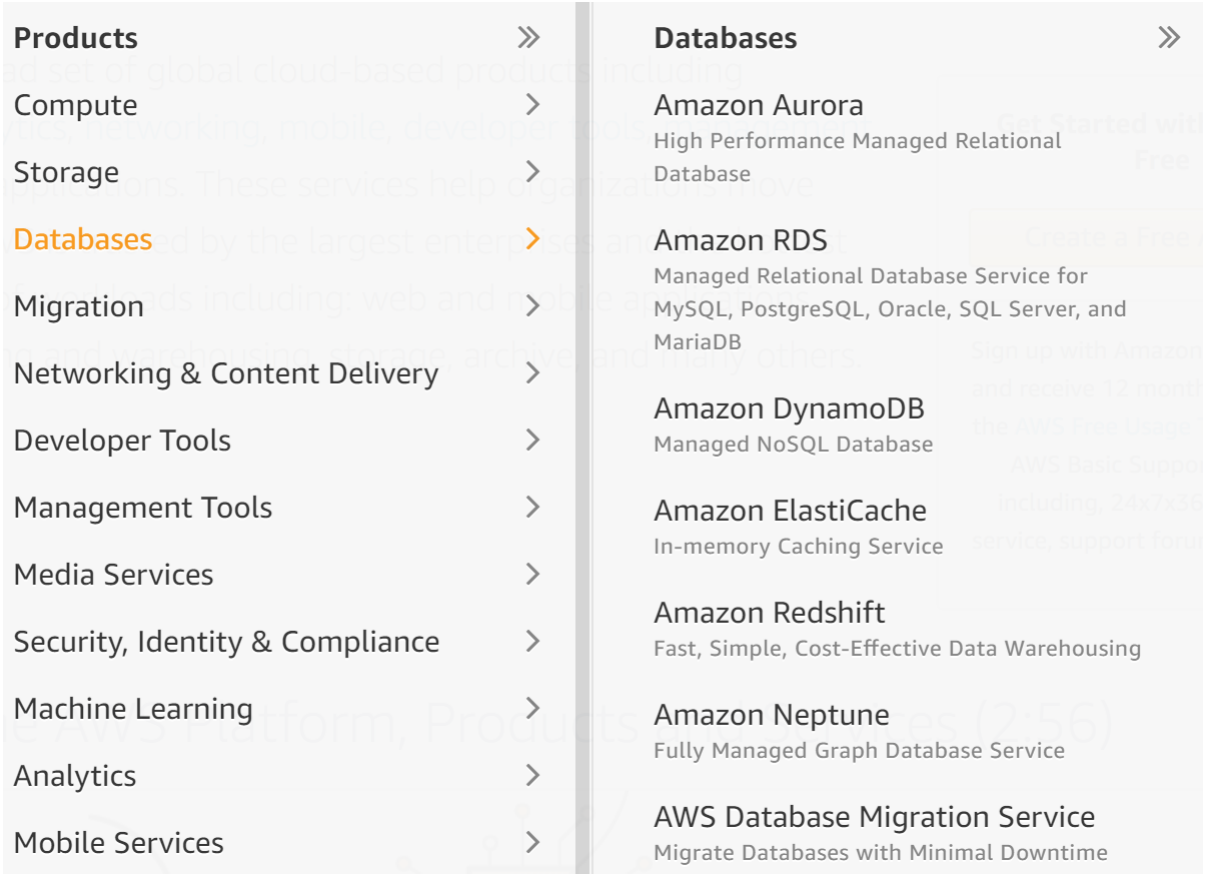
Cloud Storage and Database Services



AWS Storage Services



AWS Database Services



Storage Services

- Storage used locally by VM
  - Similar to hard disk of a computer with a file system
- Provide storage for virtual machines
  - E.g. AWS's EBS
  - Disks provided by virtual machines are ephemeral and not replicated
  - Separate storage from VM is more flexible and has higher availability
  - E.g. create an EC2 virtual machine to install MySQL, specify that the data is stored on attached EBS

Storage Services

- General storage or storage that can be used by applications
  - E.g. AWS's S3, Google Drive
- Simple Storage Service
  - Simple key value abstraction
    - Key is the address, value is the object
  - Simple put/get API
    - No random update functionality
  - Every object is addressable
    - https://s3.amazonaws.com/comp5349-data/week6/movies.csv
  - Permissions similar to file system
  - Pseudo directory structure
  - Data is replicated and can be copied in CDN servers

Cloud Database Services

- Provide distributed storage for structured or semi-structured data
  - With SQL like or subsets of SQL queries
- Most cloud service providers implement their own distributed database services to mange internal or clients' data
- Google
  - GFS(SOSP'03) => **Bigtable(OSDI'06)** => MegaStore =>Spanner
- Microsoft **Azure Storage** (**SOSP'11**, **ATC'12**)
  - Layered structure similar to GFS+Bigtable
  - Erasure coding to replace full replication
- Amazon
  - **Dynamo** (SOSP'07)=> DynamoDB => **Aurora**
- Twitter Manhattan (2014)
  - Key value storage adopting mechanisms from both Bigtable and Dynamo

Cloud Database Services Key issues

- Cluster Organization
  - Master/Slave vs Peer to Peer
- Data Partition
  - Each row has a partition key
    - How do we determine which key goes to which partition?
  - Range Partition vs Random partition
- Data Replication and consistency
  - Immutable files
    - Traditional replication
    - Erasure coding
  - Mutable files
    - Sacrifice consistency for network partition tolerance and availability
- Fault Tolerance

What are covered in this lecture

- Google's Bigtable
  - Build a random access DB layer on top of a immutable file storage layer
  - This is adopted by Microsoft Azure Storage and partially by Amazon Aurora
- Microsoft Azure Storage
  - Using Erasure Coding to minimize storage cost for immutable files to achieve the same durability and available quality of replication mechanism
  - New version of HDFS also adopts erasure coding
- Amazon Dynamo
  - Peer-to-peer network organization
  - Random partition
  - eventual consistency
- Amazon Aurora
  - Build large scale DB service with full SQL support
  - Key innovative idea is to separate DB engine layer with storage layer
    - With idea partially borrowed from Bigtable design

Outline

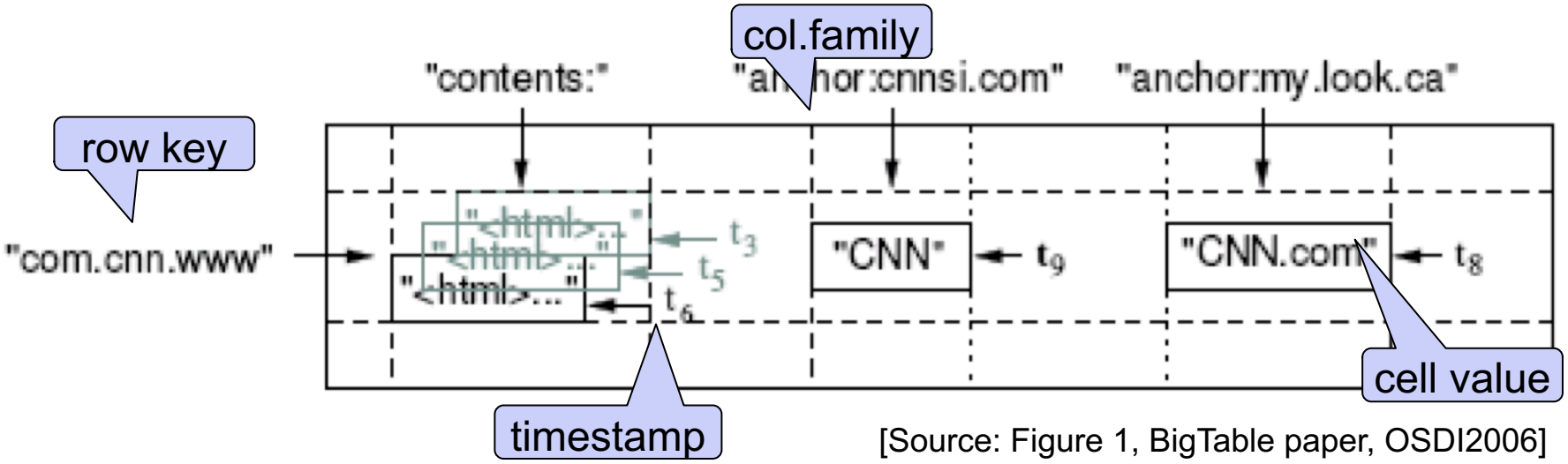
- Cloud Storage and Database Services
- **Google Bigtable**
  - **Random access service built on top of immutable file storage**
  - **Overall Architecture**
  - **Data Storage**
  - **Read/Write Path**
- Windows Azure Storage
- Amazon Dynamo
- Amazon Aurora

Bigtable

- Goals:
  - Reliable and highly scalable database for ‘Big Data’ (100s of TB – PB)
  - Designed to run on top of Google’s Distributed FS
- Initially published in OSDI 2006.
- Benefits:
  - Distributed storage (parallelism)
  - Table-like data model
  - Highly scalable
  - High availability
  - High performance

Data Model

- Basic concepts: table, row, *column family*, column, *timestamp*
  - (rowkey:string, columnKey:string, timestampe:int64) -> value: string
  - Every row has a unique *row key*
  - *Each column belongs to a column family and has a column key*
  - Cells (intersection of rows and columns) are versioned (timestamp)
  - Only data type: (uninterpreted) *byte array*



[Source: Figure 1, BigTable paper, OSDI2006]

Very Limited Query Support

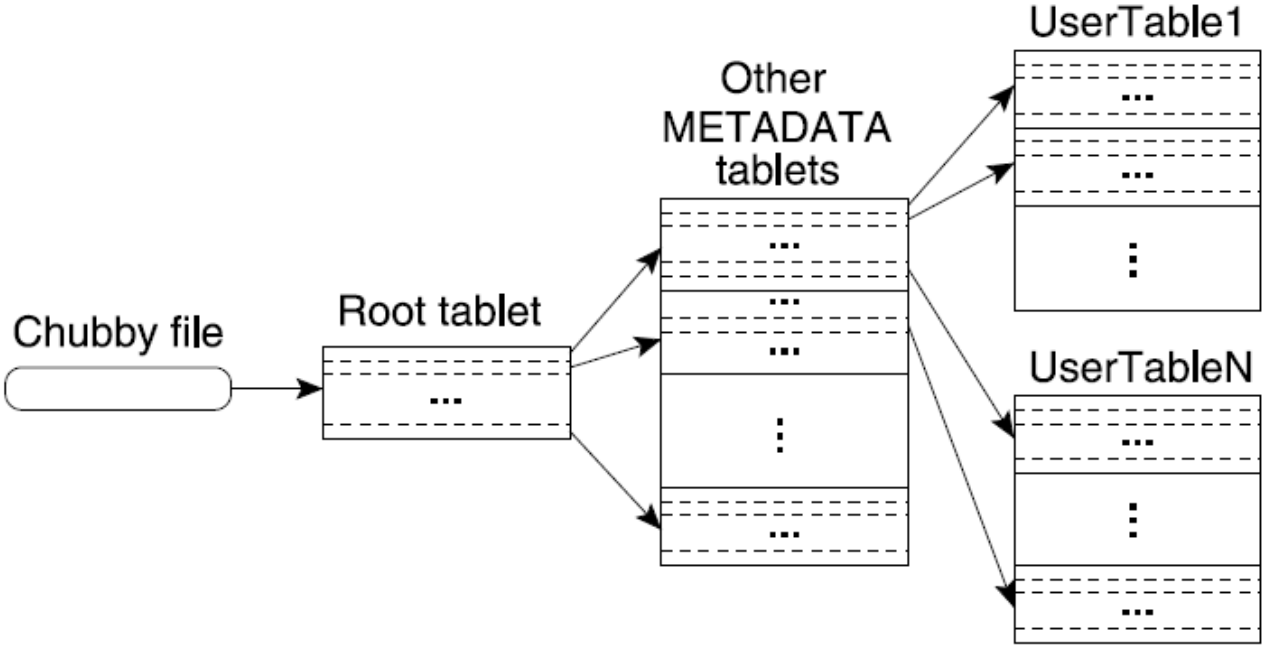
- **Get** Returns attributes for a specific **row**
- **Put** Add new rows to a table or updates existing rows.
- **Scan** Allows iteration over multiple rows for specified attributes. Scan range is a row key range
- **Delete** Removes a row from the table
- All queries are *row key* based
- Does not support querying column values

Implementation

- Master/Slave Architecture
  - A single master server
  - Many tablet Servers
- Most tables are large and are split into many small tablets
  - *Horizontal partition*: each tablet contains rows in a row key range
  - Similar to GFS(HDSF) splitting large file into chunks(blocks)
- Tablets are managed by tablet servers
  - Tablet data are stored in memory and as files (GFS files)
  - **Replication of files are handled by GFS**
- Master manages
  - Which tablet server manages which tablet
  - Load balancing tablet servers
  - Schema changes: table creation etc.

Tablet Location

- METADATA table contains the location of all tablets in the cluster
  - It might be very big and split into many tablets
- The location of METADATA tablets is kept in a root tablet
  - This can never be split
- Each tablet is assigned to **ONE** tablet server at a time.
- Both ROOT and METADATA tablets are managed by tablet servers as well

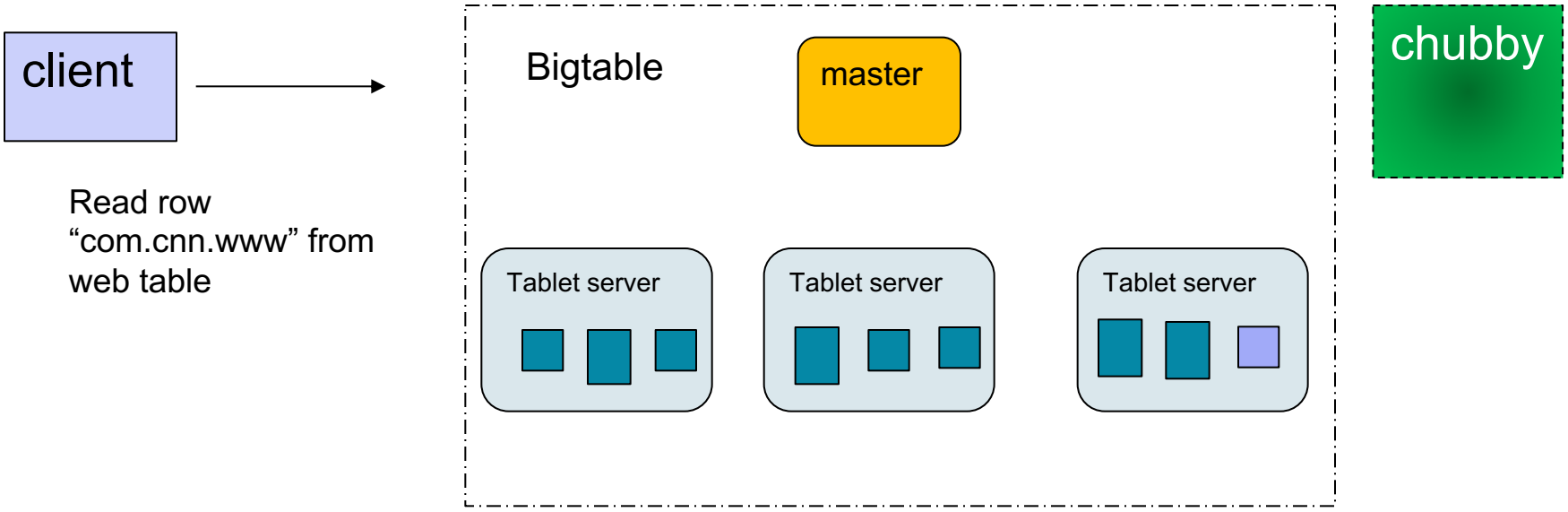


Chubby Services

- Chubby is distributed lock service consists of a small number of nodes (~5)
  - Each is a replica of one another
  - One is acting as the master
  - **Paxos** is used to select the master and also to ensure majority of the nodes have the latest data
- Chubby allows clients to create directory/file and locks on them
  - Lock has short lease time and needs to be renewed periodically
- Usage in Bigtable
  - Ensure there is only one master
  - Keep track of all tablet servers
  - Stores the root table location
  - If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable.

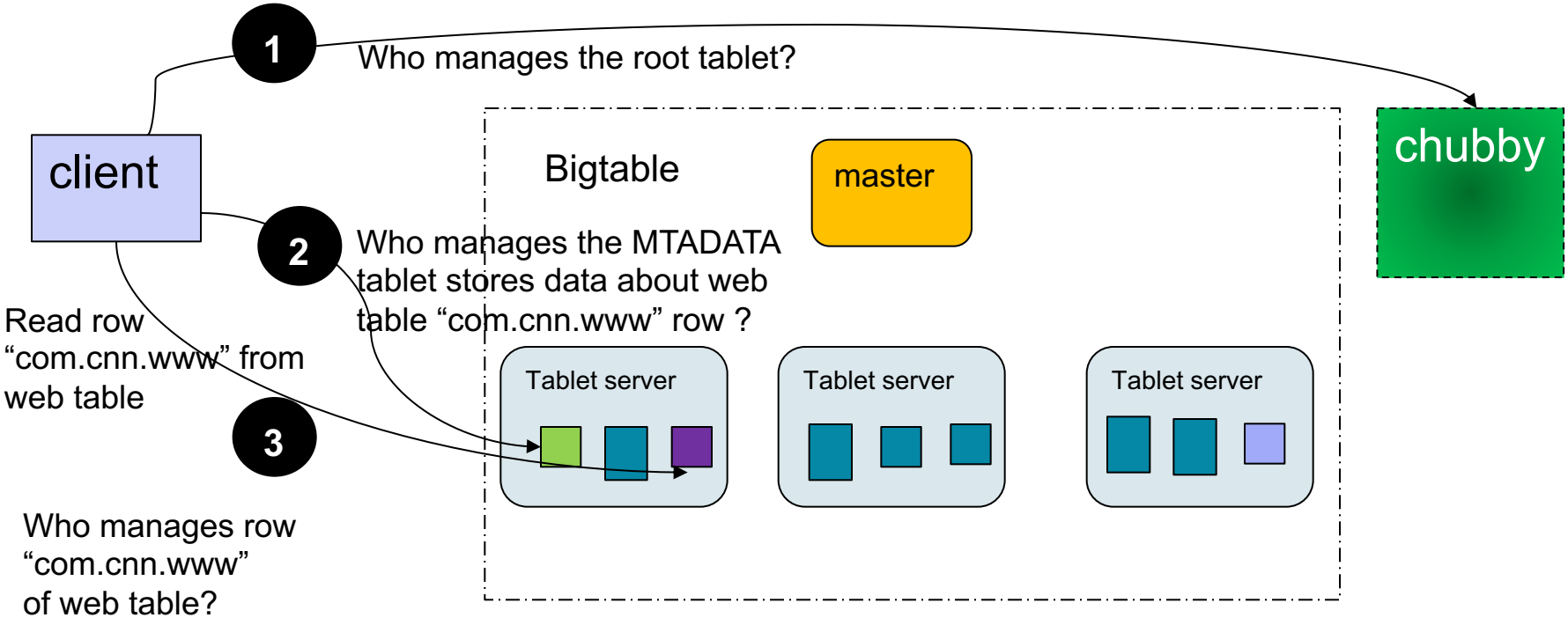
Tablet Serving

- Client read/write request
  - E.g. client wants to read the row corresponding to “com.cnn.www” from the web table
- Steps
  - Find the *tablet location*, the table server that serves the tablet
  - Contact the tablet server to perform the read/writhe request

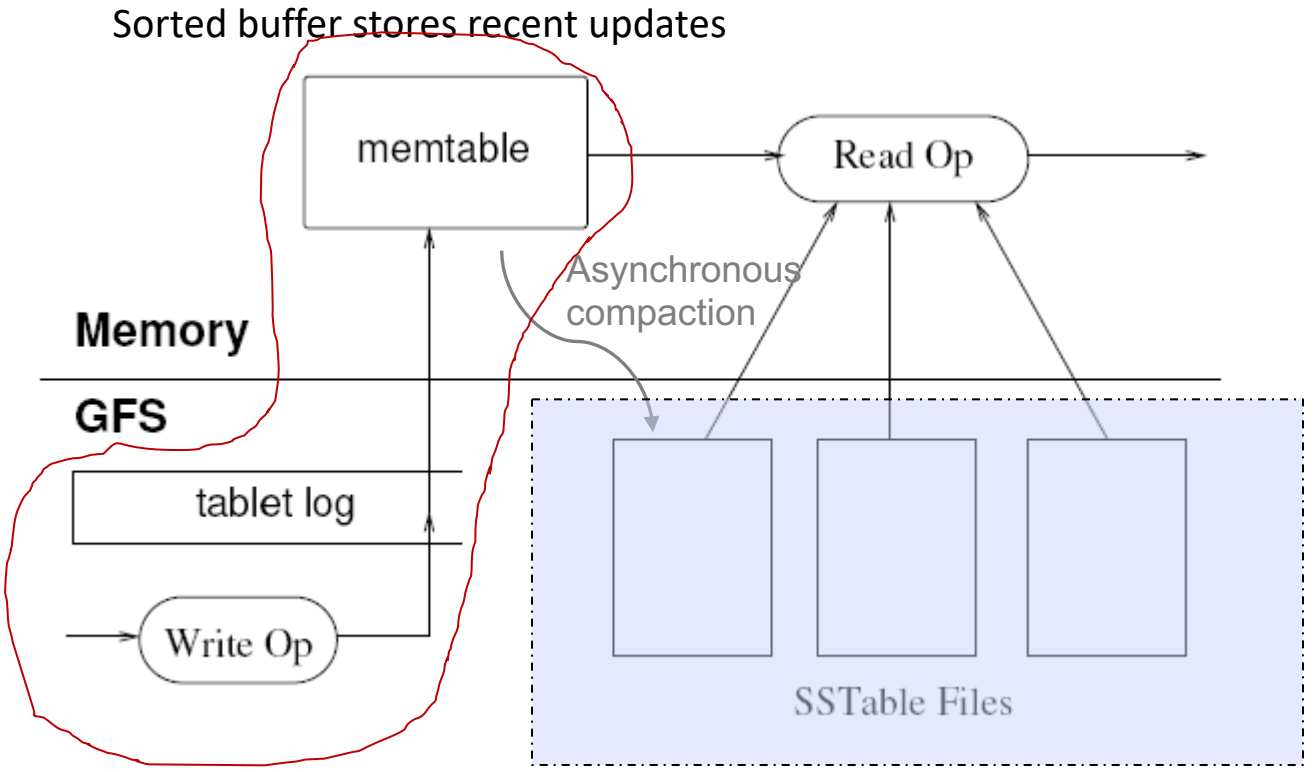


Find the tablet server

- If the client is requesting the data for first time
  - One round trip from chubby to find the root tablet's location
  - One round trip to the tablet server manages the root tablet
  - One round trip to the tablet server manages the METADATA tablet
- The client caches the tablet location for later use



Tablet Representation



Write operation

SSTable files of one tablet, which persists earlier updates

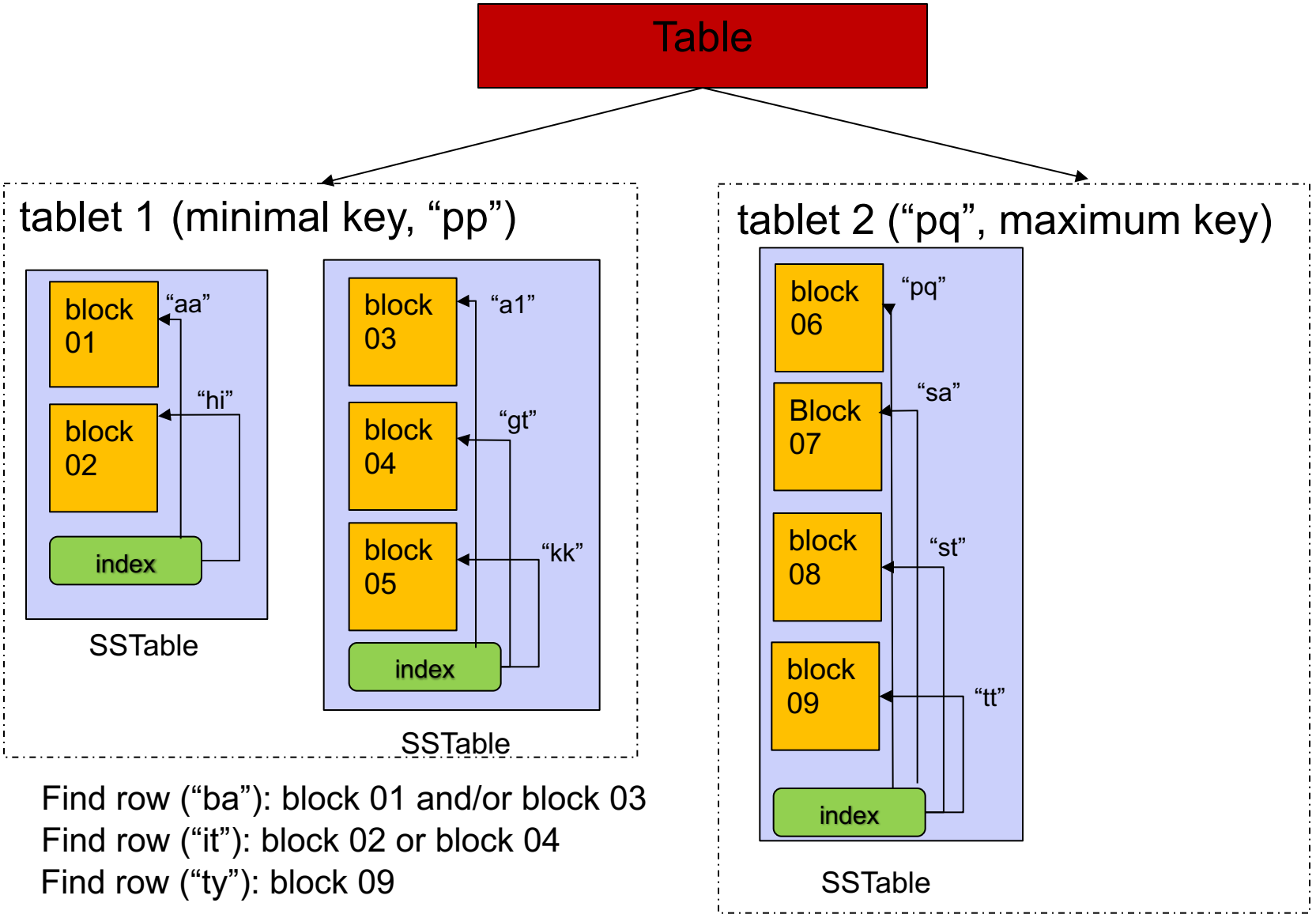
Tablet Representation Implications

- A tablet server *manages* many tablets
  - Its memory contains latest updates of those tablets (memtable)
  - BUT, the actual persisted data of those tablets might not be stored in this tablet server
    - Logs and SSTable Files are managed by the underlying file system GFS
    - GFS might replicate the files in any server
- Bigtable system is not responsible for actual file replication and placement
- The separation of concern simplifies the design

Data Storage

- Google File System (GFS)
  - Is used to store actual Bigtable data (log and data files)
  - It provides replication/fault tolerance and other useful features in a cluster environment
- Google SSTable file format
  - Bigtable data are stored internally as SSTable format
  - Each SSTable consists of
    - Blocks (default 64KB size) to store **ordered** *immutable* map of key value pairs
    - Block index
- The SSTable is stored as GFS files and are replicated

Table-Tablet-SSTable



Find row (“ba”): block 01 and/or block 03
Find row (“it”): block 02 or block 04
Find row (“ty”): block 09

Write Path

- A write operation may insert new data, update or delete existing data
- The client sends write operation directly to the tablet server
  - The operation is checked for syntax and authorization
  - The operation is written to the **commit log**
  - The actual mutation content is inserted in the **memtable**
    - Deleted data will have a special entry/marker
- The only disk operation involved in write path is to append update to commit log

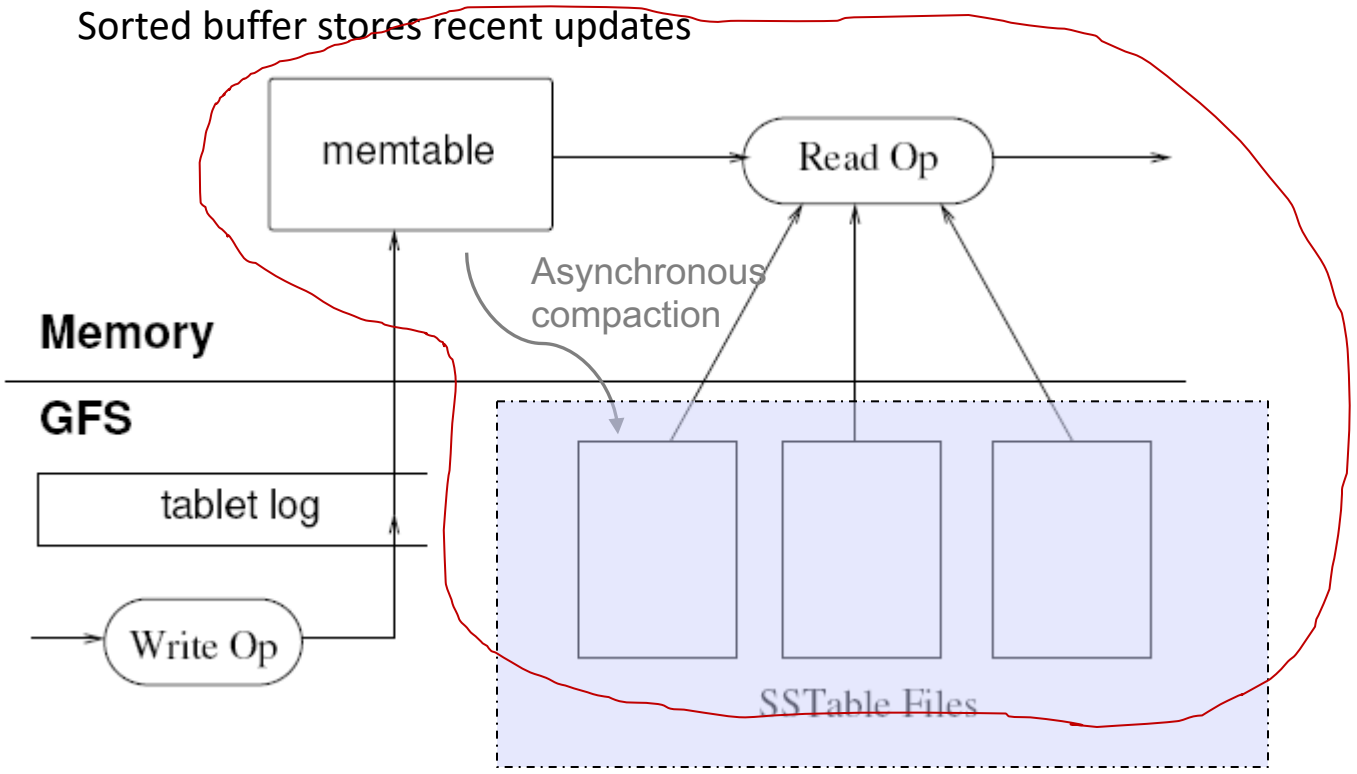
Compactions

- After many write operations, the size of memtable increases
- When memtable size reaches a threshold
  - The current one is frozen and converted to an SSTable and written to GFS
  - A new memtable is created to accept new updates
  - This is called **minor compaction**
- Why minor compaction
  - Memory management of table server
  - Reduce the size of active log entries
    - Minor compaction persists the updates on disk
    - Log entries reflecting those updates are no longer required

Compactions (cont'd)

- Every **minor compaction** creates a new SSTable
  - A tablet may contain many SSTable with overlapping key ranges
- **Merging compaction** happens periodically to merge a few SSTables and the current memtable content into a new SSTable
- **Major compaction** write all SSTable contents into a single SSTable. It will permanently remove the deleted data.

Read Path


Sorted buffer stores recent updates
memtable
Read Op
Asynchronous compaction
Memory
GFS
tablet log
Write Op
SSTable Files

Outline

- Cloud Storage and Database Services
- Google Bigtable
- **Windows Azure Storage**
  - **Erasure Coding basic principle**
  - **Classic EC**
  - **EC to minimize reconstruction cost**
- Amazon Dynamo
- Amazon Aurora

Windows Azure Storage

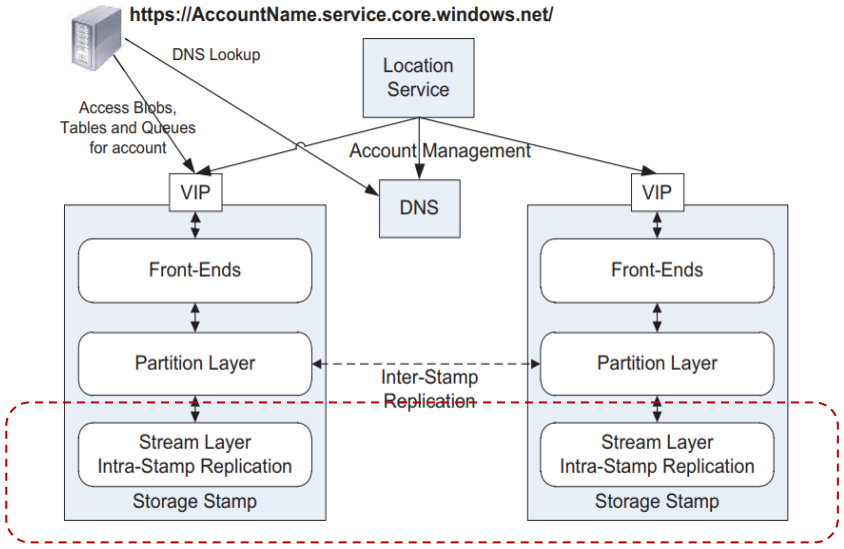


Figure 1: High-level architecture

A storage stamp is a cluster of N racks of storage nodes

Partition layer is built for managing and understanding higher level data abstractions: Blob, Table, Queue, etc

Data are partitioned and managed by partition server

The file system layer only provide internal interface for the partition layer; it is an append only file system similar to GFS

The file is called “stream”, which stores pointers to storage “blocks”. Multiple blocks form an “extent”. Extent is the unit of replication in stream layer

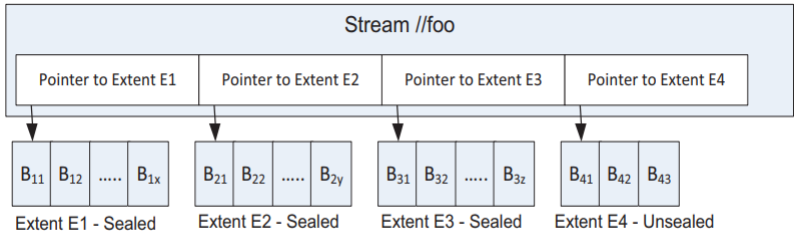
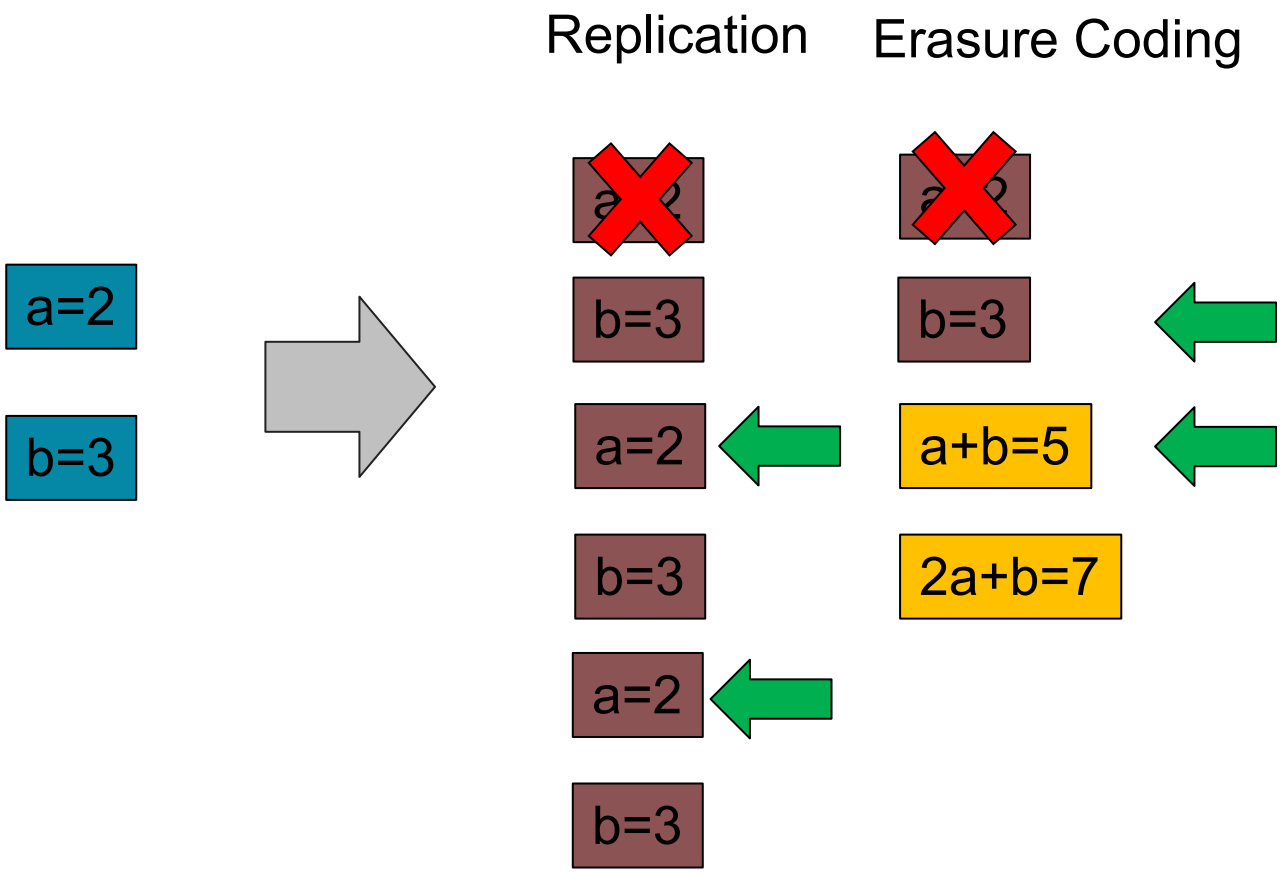


Figure 2: Example stream with four extents

Write amplification and storage overhead

- Supporting random access upper layer on top of append only file system would have write amplification problem as well as storage overhead
  - Combining new and old data usually involves creating a new file and garbage collecting all old files.
- Erasure coding is usually employed to provide sufficient ‘replication’
  - In stead of keeping 3 full copy of the data. Immutable data is divided into fragments and some parity fragments are computed and saved
  - If a fragment is lost, it can be reconstructed from the parity and the other fragments.

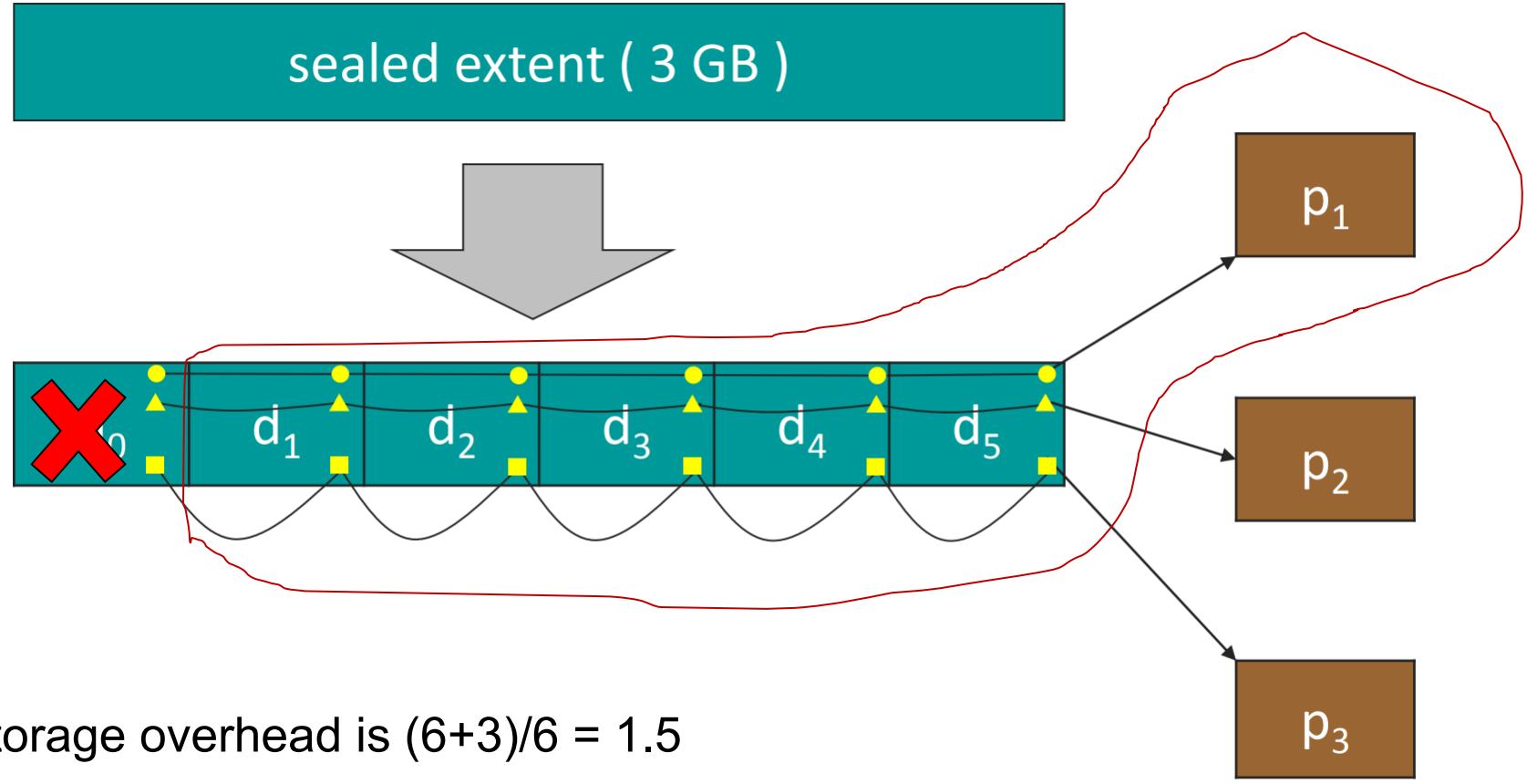
Replication vs. Erasure Coding



Trade storage cost with i/o and computation cost

Conventional Erasure Coding

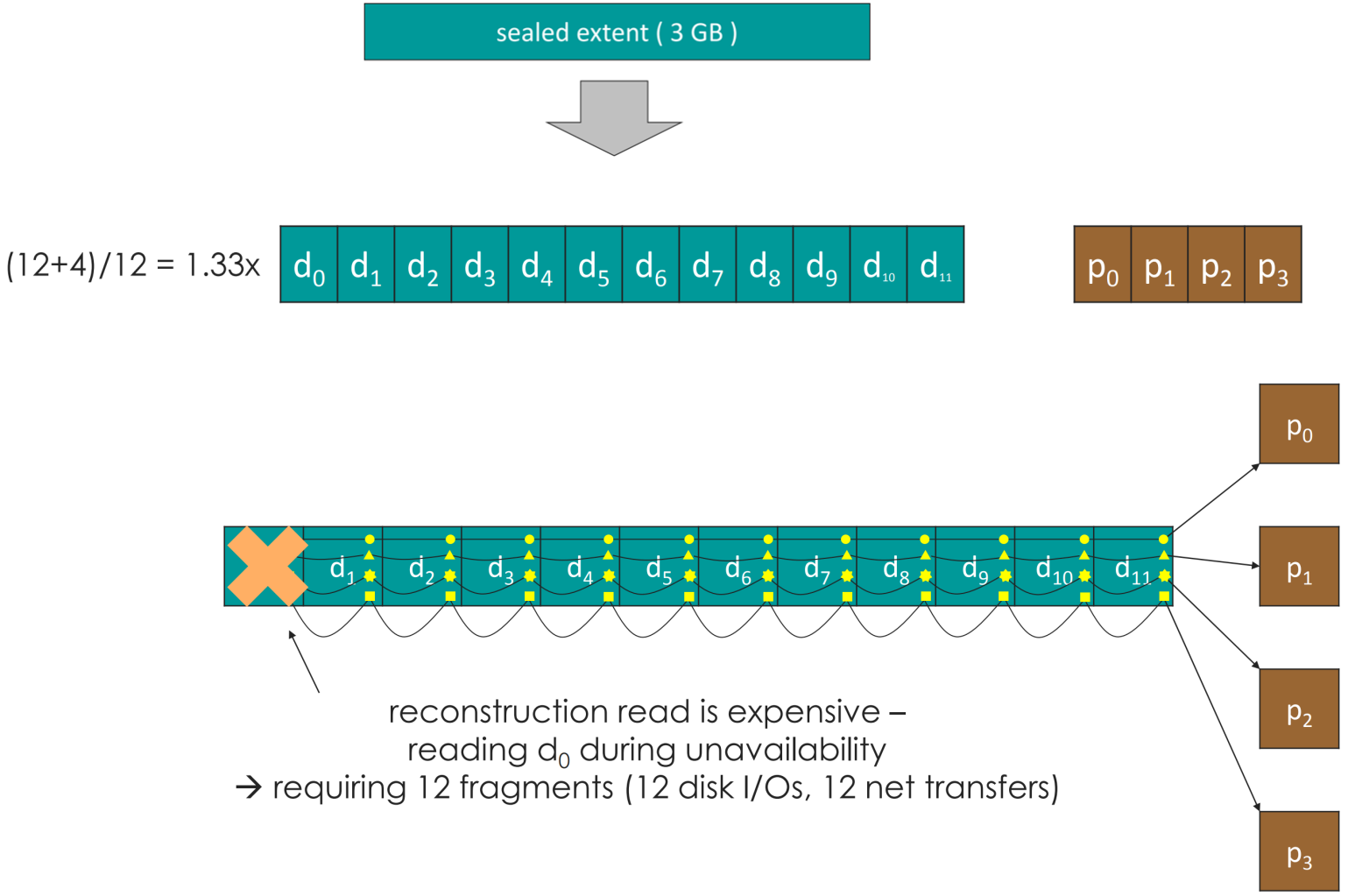
- Reed-Solomon 6+3



Storage overhead is (6+3)/6 = 1.5

Reconstruction cost involves reading 5 data fragments and 1 parity fragment

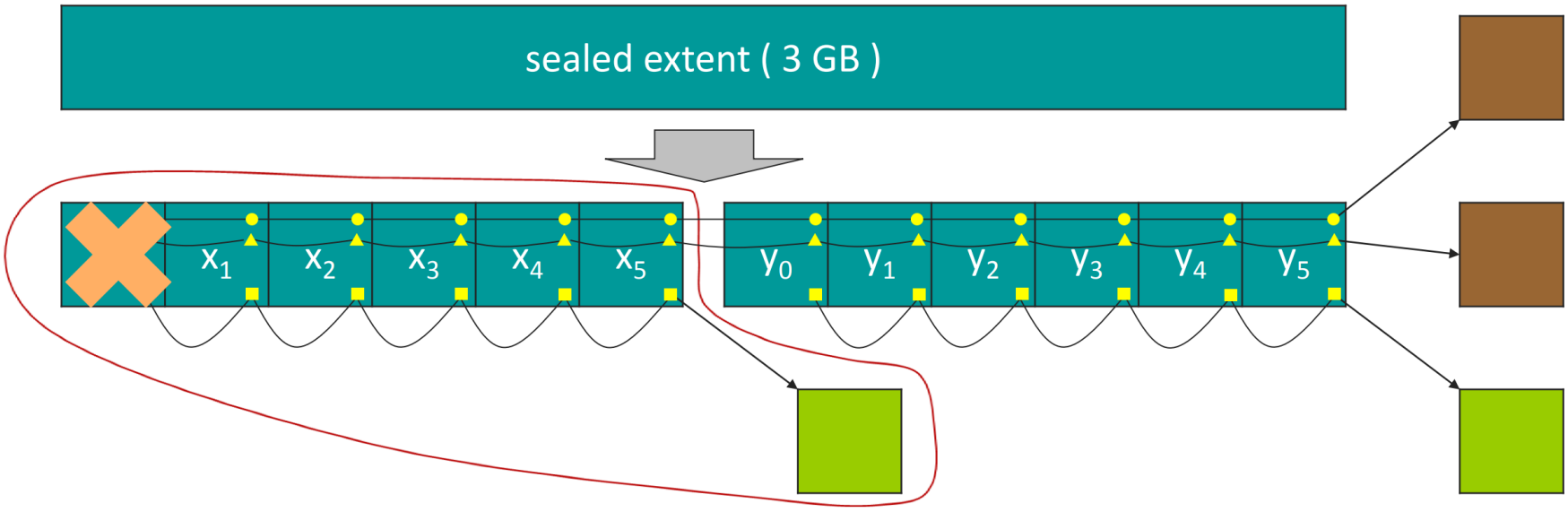
Storage Overhead vs. Reconstruction Cost



Motivation

- Achieve1.33x overhead while requiring only 6 fragments for reconstruction.
- Conventional EC considers all failures as equal
  - Cost for re-constructing one fragment equals cost of reconstructing multiple fragments
  - In cloud
    - Prob (1 failure) >> Prob (2 or more simuteneous failures)

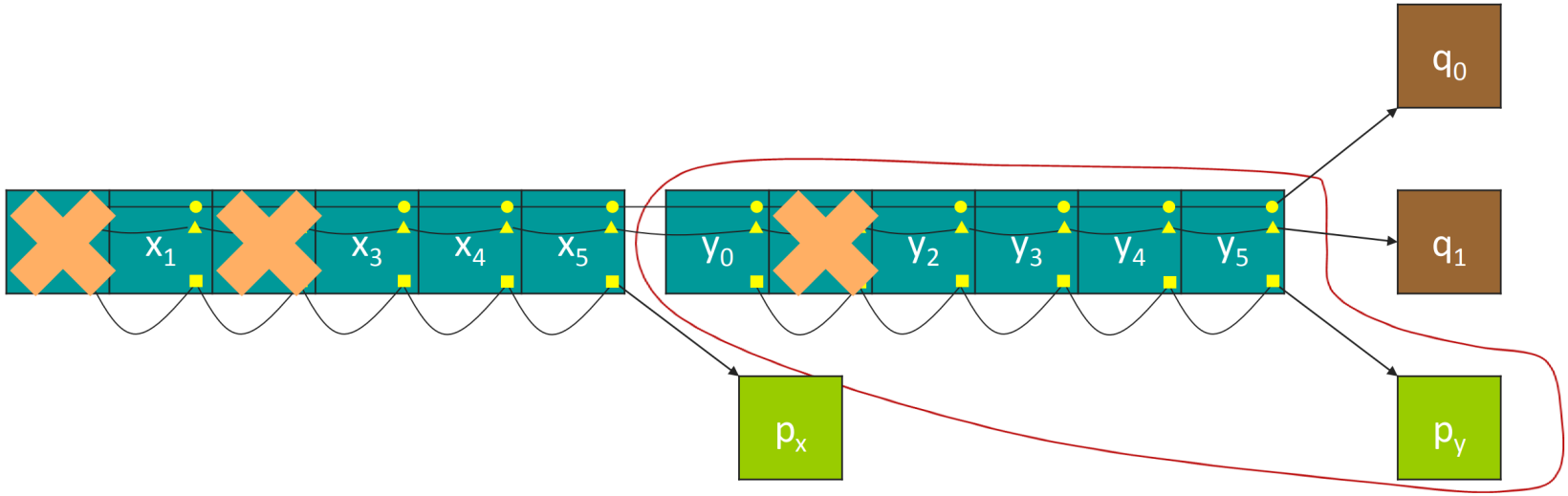
Local Reconstruction Code



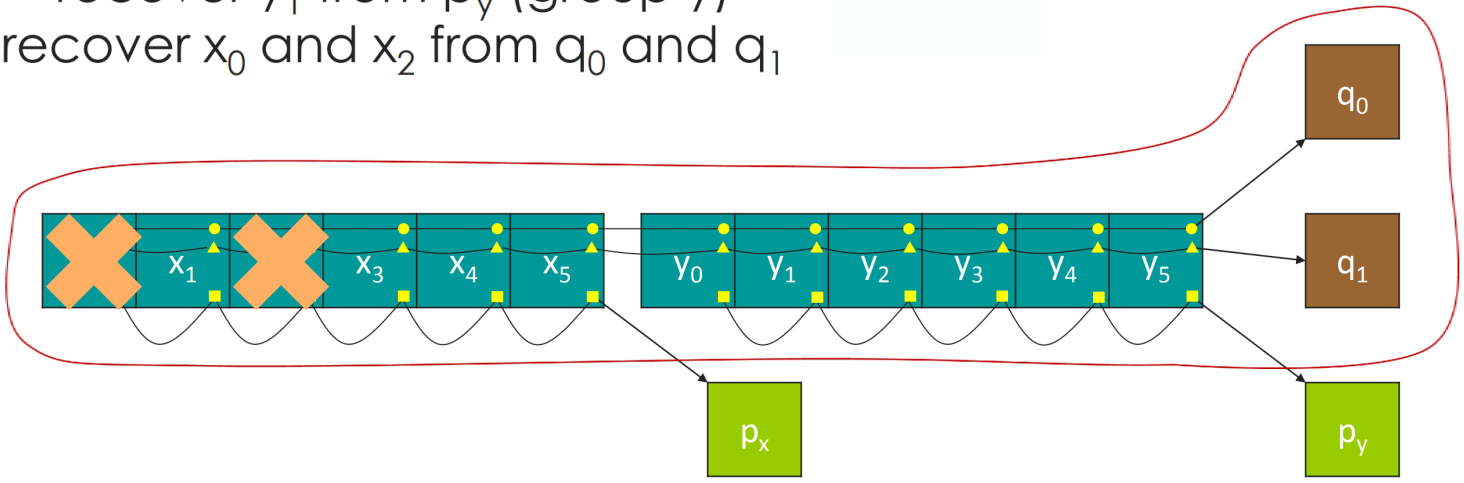
- LRC_{12+2+2}: **12** data fragments, **2** local parities and **2** global parities
  - storage overhead: (12 + 2 + 2) / 12 = 1.33x
- Local parity: reconstruction requires only 6 fragments

For single failure and reconstruction

Reconstruction multiple fragments



recover y_1 from p_y (group y)
recover x_0 and x_2 from q_0 and q_1



Outline

- Cloud Storage and Database Services
- Google Bigtable
- Windows Azure Storage
- **Amazon Dynamo**
  - **Peer-to-peer architecture**
  - **Consistent hashing**
  - **Quorum based consistency model**
- Amazon Aurora

Dynamo

- Motivation
  - Many services in Amazon only need primary key access to data store
    - E.g. shopping cart
  - Both scalability and availability are essential terms in the service level agreement
    - Always writable (write never fails)
    - Guaranteed latency
    - Highly available
- Design consideration
  - Simple key value model
  - Sacrifice strong consistency for availability
  - Conflict resolution is executed during **read** instead of write
  - Incremental scalability

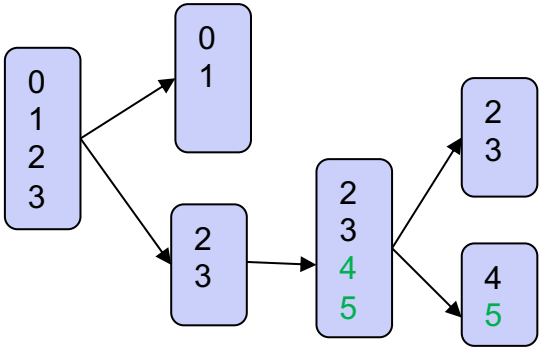
Dynamo Techniques Summary

- Dynamo is a decentralized **peer-to-peer** system
  - All nodes taking the same role
  - There is no master node

Problem	Technique	Advantage
Partitioning	**Consistent Hashing**	**Incremental Scalability**
High Availability for writes	**Vector clocks** with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	**Sloppy Quorum** and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	**Gossip-based membership** protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partitioning Algorithm

- Partition or shard a data set
  - There is a partition or shard key
    - System default vs. user specified key
  - There is an algorithm to decide which data goes to which partition
    - Range partition vs. Random(Hash) partition
- What happens when data grows
  - Bigtable way: split a partition that grows beyond threshold



- Split happens locally, does not have global impact
- Data may not be evenly distributed in each partition

Consistent Hashing

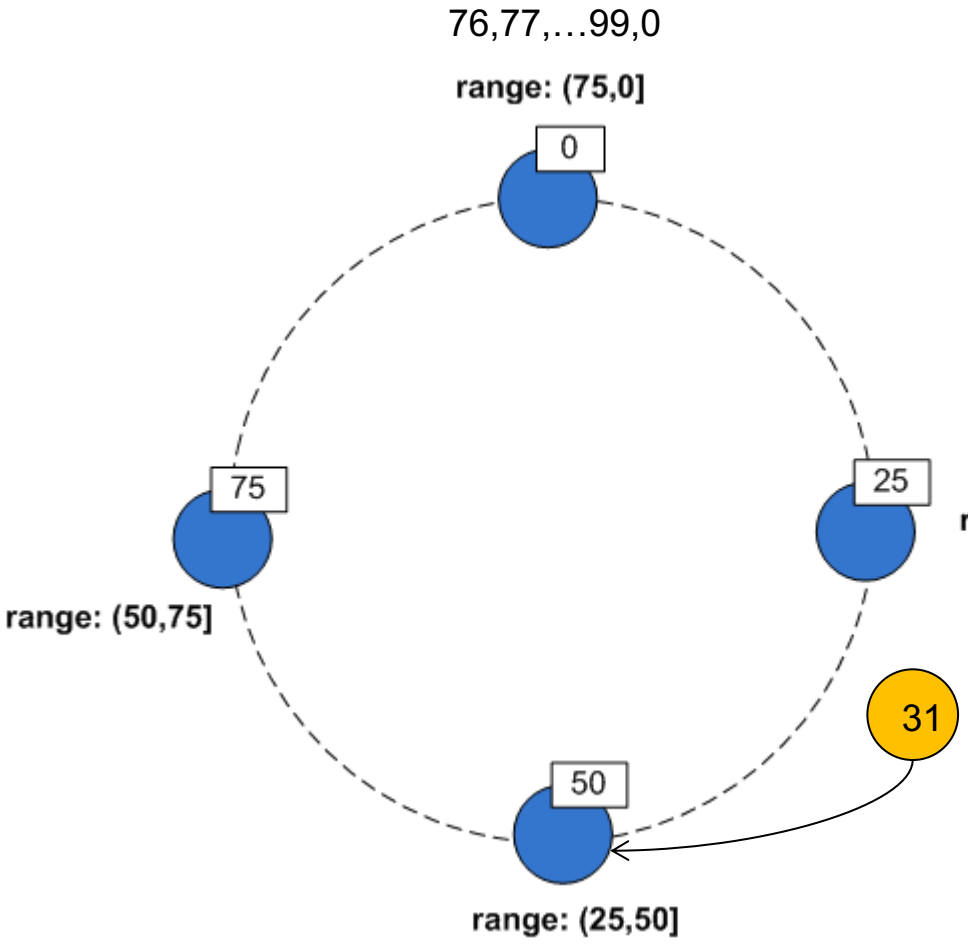
- Consistent hashing
  - " a special kind of hashing such that when a hash table is resized, only **K/n** keys need to be remapped on average, where **K** is the number of keys, and **n** is the number of slots."

    [Wikipedia: Consistent Hashing]
  - It does not identify each partition as a number in [**0,n-1**]
  - The output range of a hash function is treated as a fixed circular space or "ring" (i.e. the largest hash value wraps around to the smallest hash value).
  - Each partition represents a range in the ring space, identified by its position value (token)
  - The hash of a data record's key will uniquely locate in a range
  - In a distributed system, each node represents one partition or a number of partitions if "virtual node" is used.

Consistent Hashing

- Each node in the Dynamo cluster is assigned a “token” representing its position in the “ring”
- Each node is responsible for the region in the ring between it and its predecessor node
- The ring space is the MD5 Hash value space (128 bit)
  - 0 to 2^{127} -1
- The MD5 Hash of the key of any data record is used to determine which node is the coordinator of this key. The coordinator is responsible for
  - Storing the row data locally
  - Replicating the row data in N-1 other nodes, where N is the replication factor

Consistent Hashing Example



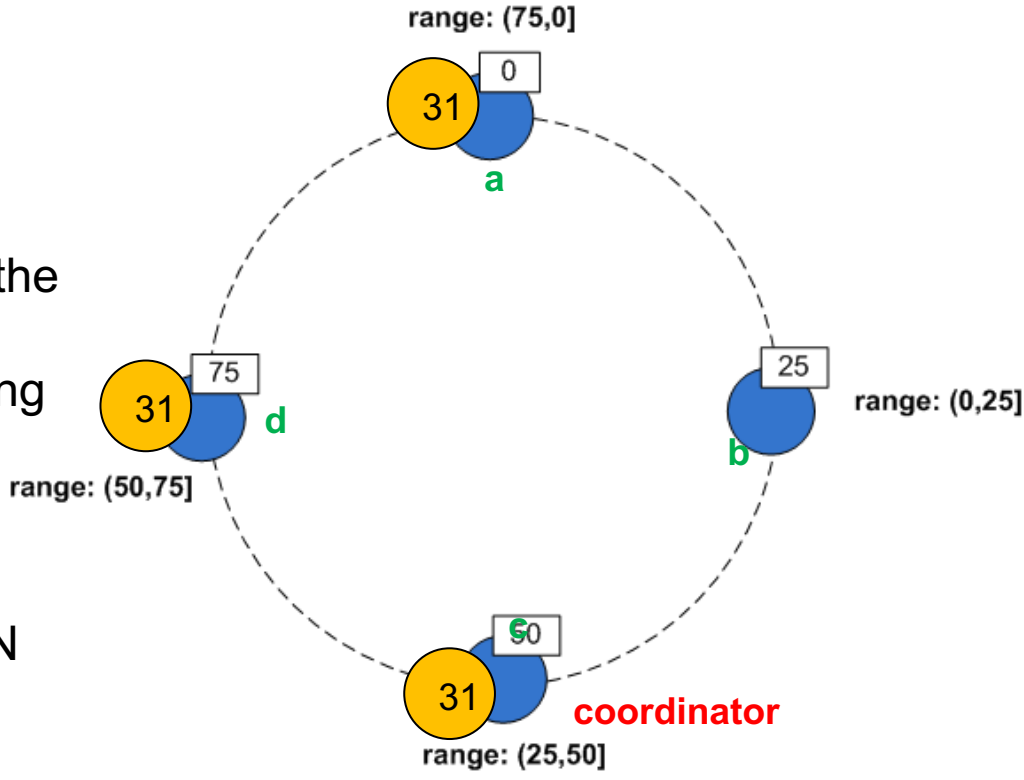
Ring space: 0~99

row key: “bsanderson”
is hashed to a number **31**

The node with token 50 is responsible for this key
This is called the ***coordinator*** of this key

Replication

- Replication is essential for high availability and durability
  - Replication factor (N)
  - Coordinator
  - Preference list
- Each key (and its data) is stored in the coordinator node as well as N-1 clockwise successor nodes in the ring
- The list of nodes that is responsible for storing a particular key is called the *preference list*
- Preference list contains more than N nodes to allow for node failures
  - Some node are used as temporary storage.
  - Can be computed on the fly by any node in the system



Preference list for this key: {c,d,a,b}

Membership and Failure Detection

- Each node in the cluster is aware of the token range handled by its peers
  - This is done using a gossip protocol
  - New node joining the cluster will randomly pick a token
  - It needs to know at least one node in the cluster
  - The information is gossiped around the cluster
- Failure detection is also achieved through gossip protocol
  - Local knowledge
  - Nodes do not have to agree on whether or not a node is “really dead”.
  - Used to handle temporary node failure to avoid communication cost during read/write
  - Permanent node departure is handled externally

Read and Write with Replication

- When there are replicas, there are many options for read/write
- In a typical Master/Slave replication environment
  - Write happens on the master and may propagate to the replica immediately and wait for all to ack before declaring success, or lazily and declare success after the master finishes write
  - Read may happen on the master (strong consistency) or at one replica (may get stale data)
- In an environment where there is no designated master/coordinator, other mechanisms need to be used to ensure certain level of consistency
  - Order of concurrent writes
  - How many replica to contact before answering/acknowledging

Dynamo Read/Write Route

- Any node is eligible to receive client read/write request
  - get (key) or put (key, value)
- The node receives the request can direct the request to the node that has the data and is available
  - Any node knows the token of other nodes
  - Any node can compute the hash value of the key in the request and the preference list
  - A node has local knowledge of node failure
- In Dynamo, “A node handling a read or write operation is known as the coordinator. Typically, this is the first among the top N nodes in the preference list.”, which is usually the coordinator of that key unless that node is not available.
  - Every node can be the coordinator of some operation
  - For a given key, the read/write is usually handled by its coordinator or one of the other top N nodes in the preference list

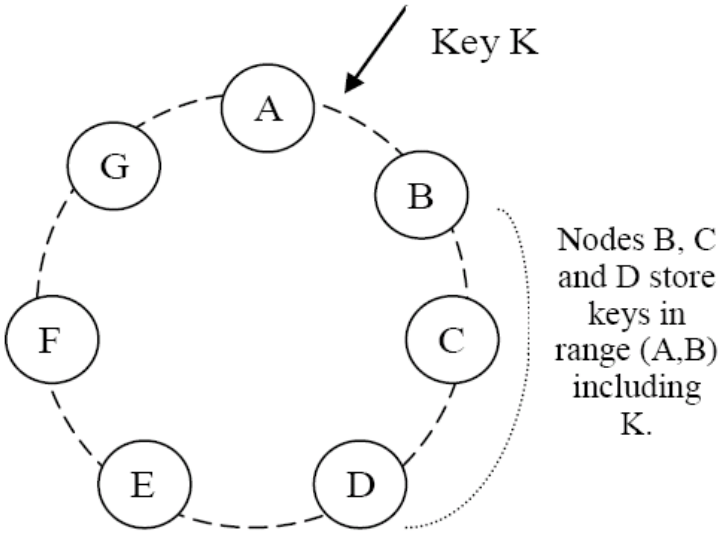
Slide based on material by Alan Fekete

Sloppy Quorum

- Quorum may include nodes that do not store a replica
  - Preference list is larger than N
  - Read/Write may have quorum members that do not overlap
- Both read and write will contact the first N *healthy* nodes from the preference list and wait for **R** or **W** responses
- Write operation will use hinted handoff mechanism if the node contacted does not store a replica of the data

Hinted Handoff

- Assume N = 3. When C is temporarily down or unreachable during a write, send replica to E.
- E is hinted that the replica belongs to C and it should deliver to C when C is recovered.
- Sloppy quorum does not guarantee that read can always return the latest value
- Assume for a subsequent read, B is down and C is back but has not received the data from E yet
  - Write set: (**B**,D,**E**)
  - Read set: (**C**,**D**,E)



Outline

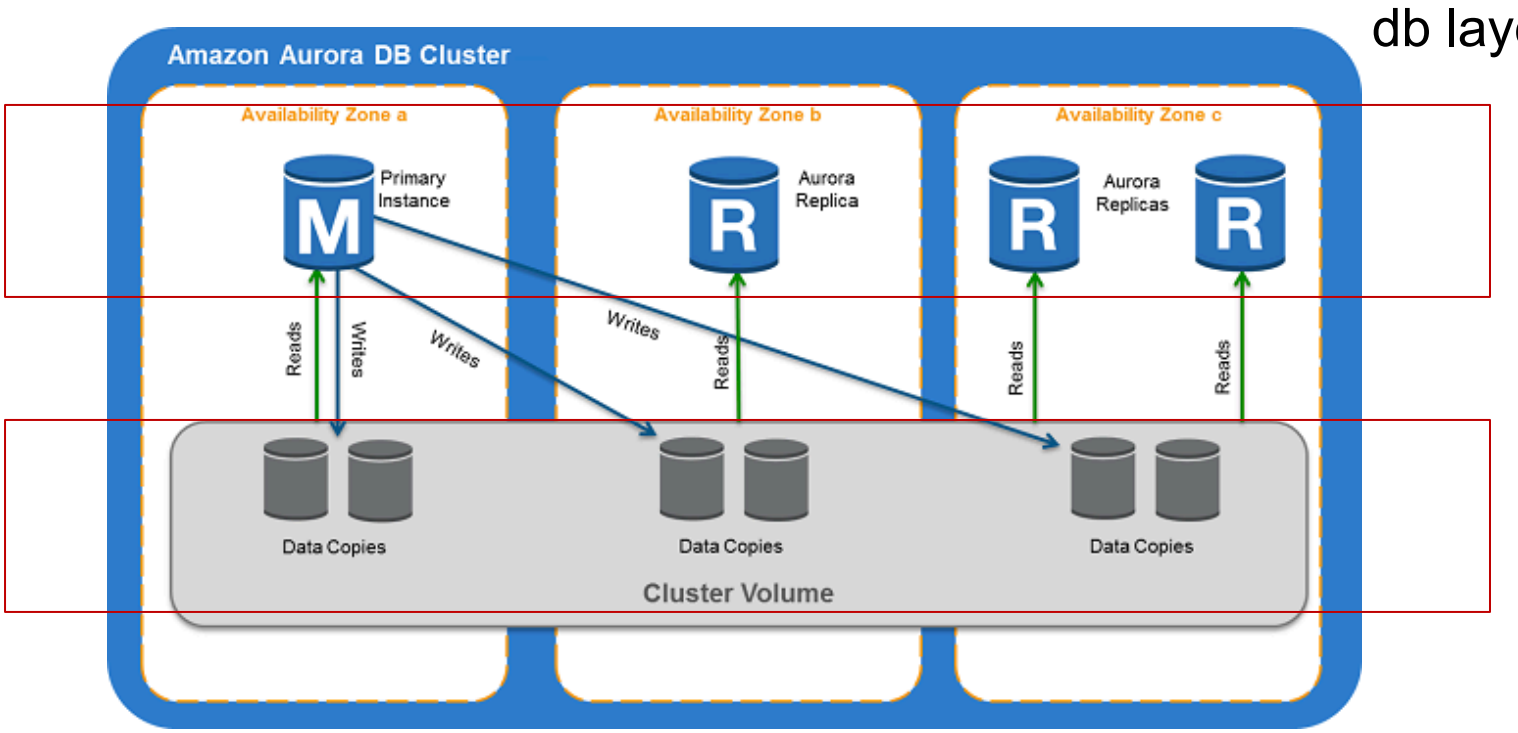
- Cloud Storage and Database Services
- Google Bigtable
- Windows Azure Storage
- Amazon Dynamo
- **Amazon Aurora**
  - DB layer vs. Storage Layer
  - Log as database

Amazon Aurora

- Amazon Aurora is a relational Database service attempting to build SQL in a distributed environment
  - It runs on modified MySQL/PostgresSQL
  - With many storage layer optimization
- The idea is to offer a service with full SQL support but much more scalable
  - It also comes with many other desirable features such as durability, fault tolerance, high availability

Aurora Overview

- Key design principles of Aurora
  - Separate DB engine layer from storage layer
    - Think about GFS+Bigtable, WAS, etc



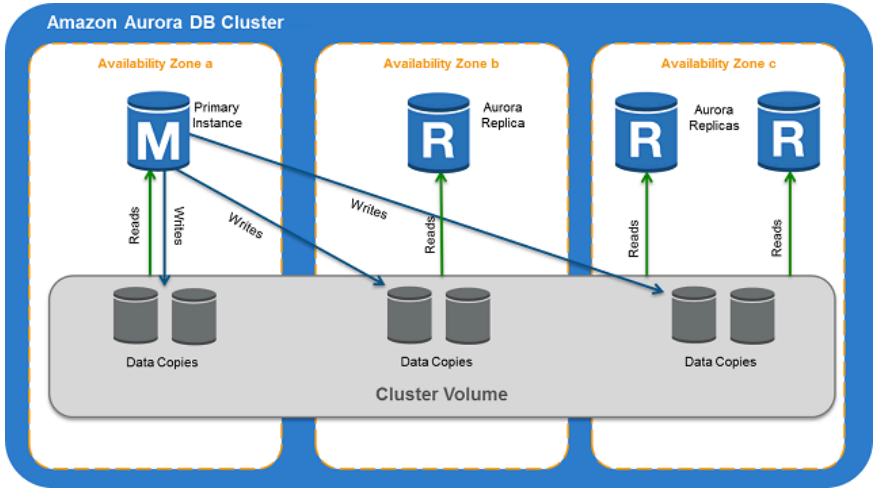
https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Aurora.Overview.html

DB Layer

- From client perspective, the DB layer is quite similar to mirrored MySQL configuration
  - Primary (writer) and replicas (readers) of DB
    - Run modified MySQL/PostgresSQL
  - But the actual data is **not stored** on the instance that runs DB engine
    - In fact, Aurora support much larger data volume than traditional RDBMS system can support: 64 TB at the publishing time
    - It can support 1 writer + 15 reader
  - Latest update will be reflected in the **memory** of all available DB instances

Storage Layer

- Storage nodes
  - A group of EC2 instances with local SSD
  - They are scattered in different AZs
  - They persist and *replicate* the data for the database.
  - The database volume is "divided into small fixed size segments, currently 10GB in size. These are each replicated 6 ways into Protection Groups (PGs) so that each PG consists of six 10GB segments, organized across three AZs, with two segments in each AZ."



Does not mean only 6 storage nodes

100GB database will have 10 PG groups and could be replicated in many storage nodes.

Durability at Scale

- Replication is a typical way to achieve *durability* as well as *availability* and *scalability*
- Typical replication factor in most distributed storage system is three
  - A quorum mechanism is used to ensure data consistency
  - W=2, r=2
- In large scale distributed system with multiple availability zones
  - Replicas are usually distributed in different AZs to minimize the chance of concurrent failures
- Aurora double the count of replicas in each AZ to tolerate
  - Losing an entire AZ and one additional node without losing data
  - Losing an entire AZ without impacting the write ability
- Data has 6 copies in 3 AZ and uses a write quorum of 4 and read quorum of 3

The Burden of Amplified Write

- To answer a write request, a database engine usually needs to
  - Append the Write-ahead log(WAL)
  - Update the actual data file
- A number of other data must also be written in practice
  - Binary log
  - Double-write
  - Metadata file
  - Etc..

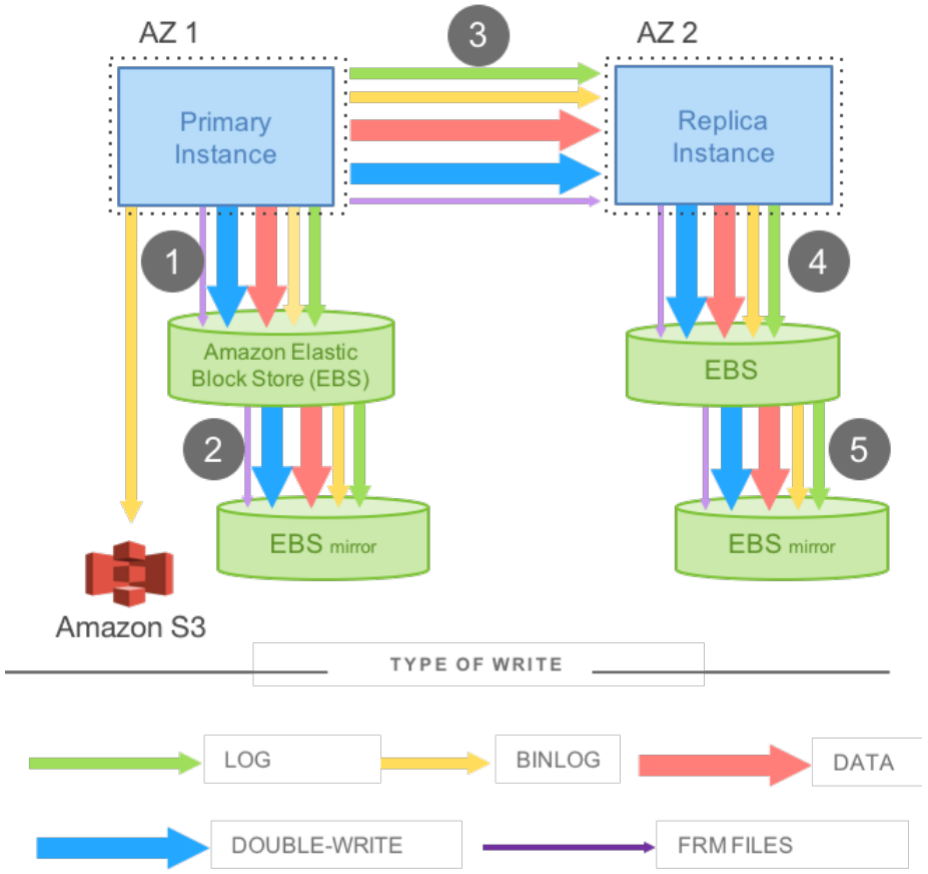


Figure 2: Network IO in mirrored MySQL

Aurora Approach: The Log is the Database

- During write operation
  - Only redo logs are sent across the network by **Primary Instance**
- **Replicate Instances** receive redo logs to update their memory
- **Storage nodes** receive redo logs
  - They materialize database pages from logs independently and in asynchronous manner
- **Primary Instance** waits for 4 out of 6 acknowledgements to consider the write as successful.

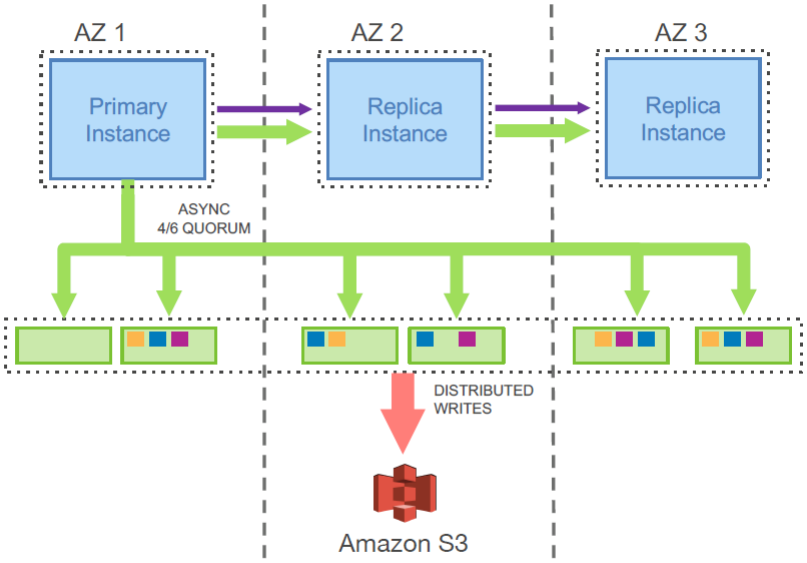


Figure 3: Network IO in Amazon Aurora

IO traffic in Aurora Storage Nodes

- **(1)** receive log record and add to an in-memory queue
- **(2)** persist record on disk and acknowledge
- (3) organize records and identify gaps in the log since some batches may be lost,
- (4) gossip with peers to fill in gaps,
- (5) coalesce log records into new data pages
- (6) periodically stage log and new pages to S3, (7) periodically garbage collect old versions, and finally (8) periodically validate CRC codes on pages

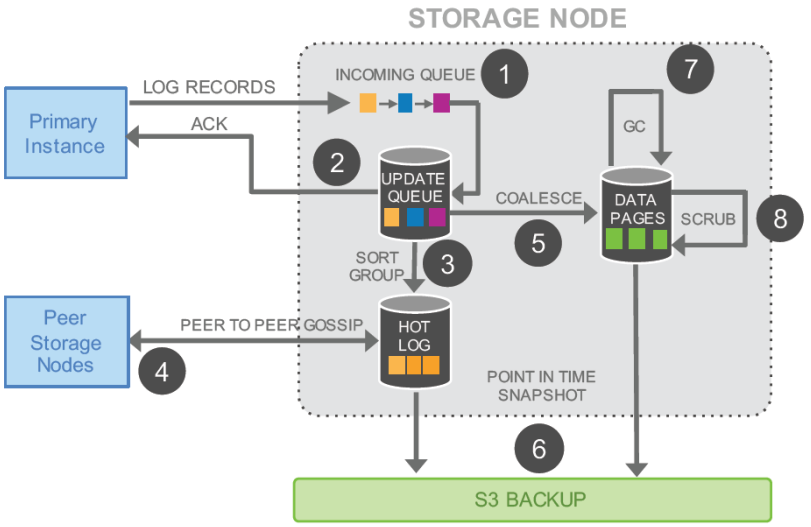


Figure 4: IO Traffic in Aurora Storage Nodes

Design Consideration: Minimize the latency of the **foreground** write request

Implementation: Majority of the storage processing is moved to **background**

The Log Marching Forward

- Log advances as an ordered sequence of changes
- Each log record has an associated Log Sequence Number(LSN) that is a monotonically increasing value generated by the *database primary instance*.
- DB primary instance handles multiple transactions at any particular time point
  - Operations in the transactions are either all executed or 0 executed
  - Log record of last operion in a transaction represents a possible consistency point
    - DB marks a few such points at any time
- Storage service might have different storage status at any time
  - DB: I have issued logs for transaction x, y, z,…
  - SS: We have not received all logs of transaction x yet, but have received all for transaction y
  - If DB crashes at this point, at recovery time, it needs to know transaction x should be aborted and all persisted logs should be disscarded.

Combining DB view and Storage View

- During crash recovery, the storage service determines the highest LSN for which it can guarantee availability of all prior log records: ***VCL*** (Volume Complete LSN)
  - Lower level view, reflecting storage service status
- The database layer can specify a few particular LSNs as Consistency Point LSN (***CPL***)
  - Higher level view, DB can establish it based on transaction log record
- ***VDL***(Volume Durable LSN) is the highest CPL that is smaller than or equal to VCL

Write/Read Path

- During normal *write* operation
  - Storage nodes gossip with peers to fill in the gaps of missing logs after it acknowledges the write request. The write request is considered as successful after the write quorum is satisfied
  - Database can advance its VDLs
- During normal read operation
  - DB engine will search its memory first and may return the result without any disk IO involved, that is, storage services may not be involved
  - When there is a miss, it needs to read data from storage nodes.
  - Read quorum is not required to establish consensus
    - DB specify a read-point represented by VDL
    - It then selects a storage node that is *complete* with respect to VDL, meaning the node has received all logs prior to the VDL

References

- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, **Bigtable: A Distributed Storage System for Structured Data, OSDI**'06: In Proceedings of the Seventh Symposium on Operating System Design and Implementation (OSDI'06), Seattle, WA, 2006
- Calder, Brad, et al. "Windows Azure Storage: a highly available cloud storage service with strong consistency." Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles. ACM, 2011.
- Huang, Cheng, et al. "Erasure coding in windows azure storage." Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12). 2012.
- Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. **Dynamo: amazon's highly available key-value store**. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (SOSP '07). 205-220.
- Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017**. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases.** In Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17). ACM, New York, NY, USA, 1041-1052. DOI: https://doi.org/10.1145/3035918.3056101