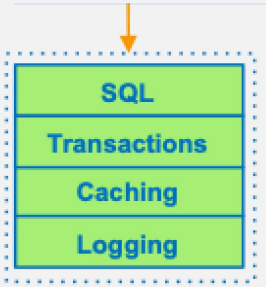
# Relational Database on the Cloud

- Traditional RDBMS rely on B+Trees, replication, etc. to optimize usage on one or a few servers
- Cloud brings many new things to the table
  - Backend storage
  - Network
  - Worldwide Scalability
- How can we optimize RDBMS for the cloud?
- First step: separate storage layer from the transactional logic
  - Decoupling storage from compute
- Deuteronomy
  - Transaction Component (TC) provides concurrency control and recovery
  - Data Component (DC) provides access methods on top of LLAMA, a latch-free log-structured cache and storage manager.
- Aurora, CosmosDB both inspired by Deuteronomy

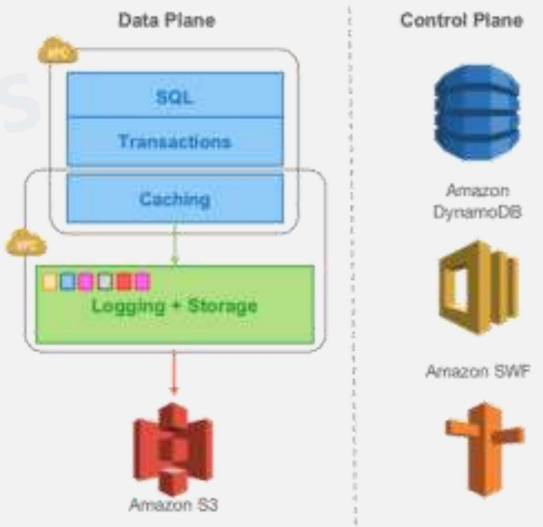
# Amazon AWS Aurora

- Optimized DB engine, built from MySQL (and later PostgreSQL), with a distributed storage layer
- API compatible with MySQL or Postgres
  - i.e. you can use an existing application
- Separate storage and compute
  - Query processing, transactions, concurrency, buffer cache, and access
  - Logging, storage, and recovery that are implemented as a scale out service.
- Move caching and logging layers into a purpose-built, scale-out, self-healing, multitenant, database-optimized storage service



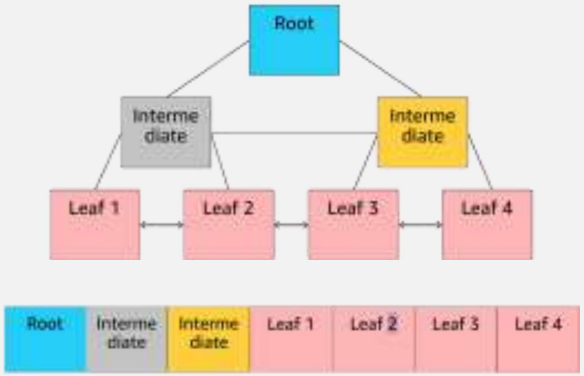
# Amazon AWS Aurora

- Each instance still includes most of the components of a traditional kernel (query processor, transactions, locking, buffer cache, access methods and undo management)
- Several functions (redo logging, durable storage, crash recovery, and backup/restore) are off-loaded to the storage service



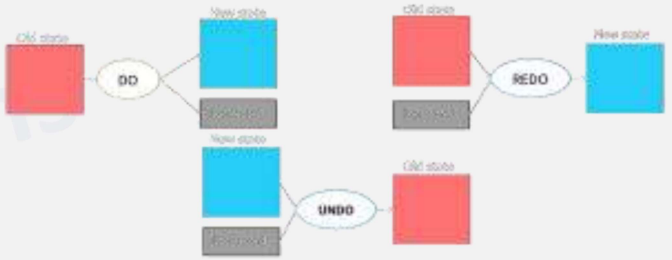
# Redo Logging

- Traditional relational databases organize data in *pages* (e.g. 16KB), and as pages are modified, they must be periodically flushed to disk
  - B+ Tree
- For resilience against failures and maintenance of ACID semantics, page modifications are also recorded in *do-redo-undo log records*, which are written to disk in a continuous stream.
- rife with inefficiencies.
  - E.g. a single logical database write turns into multiple (up to five) physical disk writes, resulting in performance problems.
  - Write Amplification
  - combat the write amplification problem by reducing the frequency of page flushes
  - This in turn worsens the problem of crash recovery duration



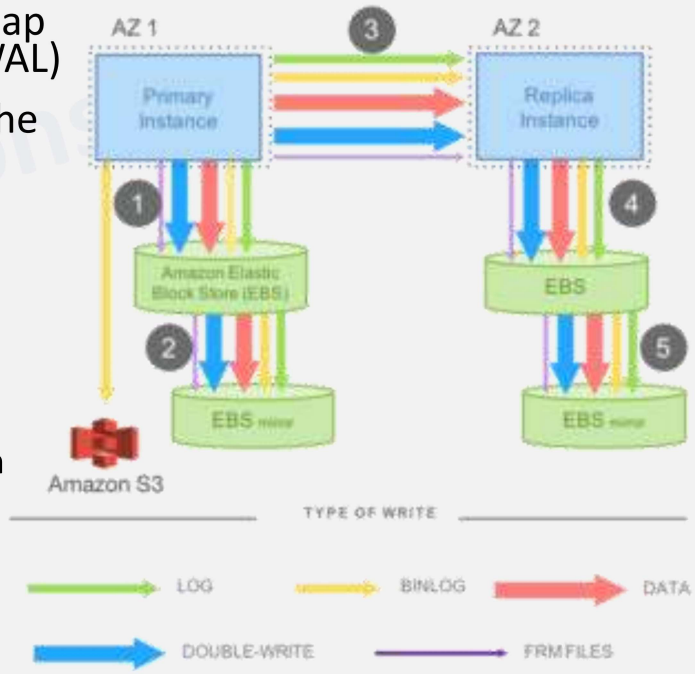
# Redo Logging

- Traditional relational databases organize data in *pages* (e.g. 16KB), and as pages are modified, they must be periodically flushed to disk
  - B+ Tree
- For resilience against failures and maintenance of ACID semantics, page modifications are also recorded in *do-redo-undo log records*, which are written to disk in a continuous stream
  - redo log record : difference between the after and the before-image of a page
- rife with inefficiencies.
  - E.g. a single logical database write turns into multiple (up to five) physical disk writes, resulting in performance problems.
    - Write Amplification
  - Combat the write amplification problem by reducing the frequency of page flushes
    - This in turn worsens the problem of crash recovery duration



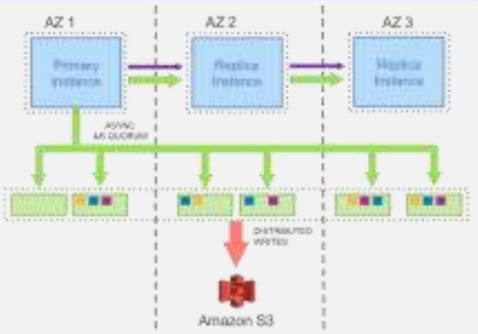
# Burden of Amplified Writes

- A system like MySQL writes data pages to objects it exposes (e.g., heap files, b-trees etc.) as well as redo log records to a write-ahead log (WAL)
- The writes made to the primary EBS volume are synchronized with the standby EBS volume using software mirroring
- Data needed to be written in step 1:
  - redo log (typically a few bytes, transaction commit requires the log to be written)
  - binary (statement) log that is archived to Amazon S3 to support point-in-time restores
  - modified data pages (the data page write may be deferred, e.g. 16KB)
  - a second temporary write of the data page (double-write) to prevent torn pages (e.g. 16KB)
  - metadata (FRM) files
- Steps 1, 3, and 5 are sequential and synchronous
  - Latency is additive because many writes are sequential
  - 4/4 write quorum requirement and is vulnerable to failures and outlier performance
- Different writes representing the same information in multiple ways



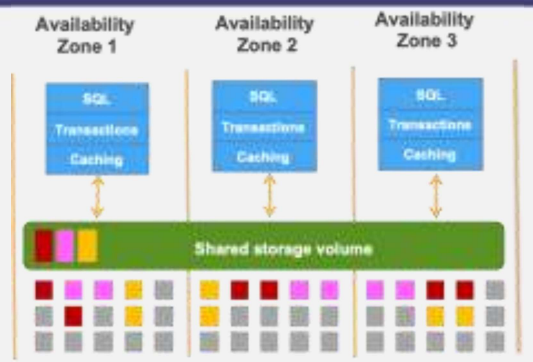
# Log is the database

- In Amazon Aurora, the log is the database
- Database instances write redo log records to the distributed storage layer, and the storage takes care of constructing page images from log records on demands from the database
  - Write performance is improved due to the elimination of write amplification and the use of a scale-out storage fleet
  - 5x write IOPS on the SysBench benchmark compared to Amazon RDS for MySQL running on similar hardware
  - Database crash recovery time is cut down, since a database instance no longer has to perform a redo log stream replay
    - From 27 seconds down to 7 seconds according to one benchmark



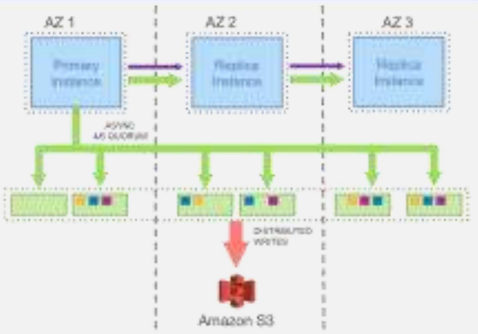
# Aurora Replication and Quorum

- Everything fails all the time
  - The traditional approaches of blocking I/O processing until a failover can be carried out—and operating in "degraded mode" until recovery —are problematic at scale
    - in a large system, the probability of operating in degraded mode approaches 1
- Aurora uses quorums to combat the problems of component failures and performance degradation
  - Write to as many replicas as appropriate to ensure that a quorum read always finds the latest data
- Goal is Availability Zone+1: tolerate a loss of a zone plus one more failure without any data durability loss, and with a minimal impact on data availability
  - 4/6 quorum
  - For each logical log write, issue six physical replica writes
    - Write operation successful when four of those writes complete
    - Instances only write redo log records to storage
    - Typically 10s to 100s of bytes, makes a 4/6 write quorum possible without overloading the network
- If a zone goes down and an additional failure occurs, can still achieve read quorum (3/6), and then quickly regain the ability to write by doing a *fast repair*



# Offloading Redo Processing to Storage

- In Aurora, the only writes that cross the network are redo log records
  - No pages are ever written from the database tier, not for background writes, not for checkpointing, and not for cache eviction
- log applicator is pushed to the storage tier to generate database pages in background or on demand
  - Generating each page from the complete chain of its modifications from the beginning of time is prohibitively expensive
  - Continually materialize database pages in the background to avoid regenerating them from scratch on demand every time
- The storage service can scale out I/Os in an embarrassingly parallel fashion without impacting write throughput of the database engine
- primary only writes log records to the storage service and streams those log records as well as metadata updates to the replica instances
- database engine waits for acknowledgements from 4 out of 6 replicas in order to satisfy the write quorum



# Aurora Fast Repair

- Amazon Aurora approach to replication: based on sharding and scale-out architecture
- An Aurora database volume is logically divided into 10-GiB logical units (*protection groups*), and each protection group is replicated six ways into physical units (*segments*)
- When a failure takes out a segment, the repair of a single protection group only requires moving ~10 GiB of data, which is done in seconds.
- When multiple protection groups must be repaired, the entire storage fleet participates in the repair process.
  - Massive bandwidth to complete the entire batch of repairs
- A zone loss followed by another component failure → Aurora may lose write quorum for a few seconds for a given protection group
  - Recovery is quick

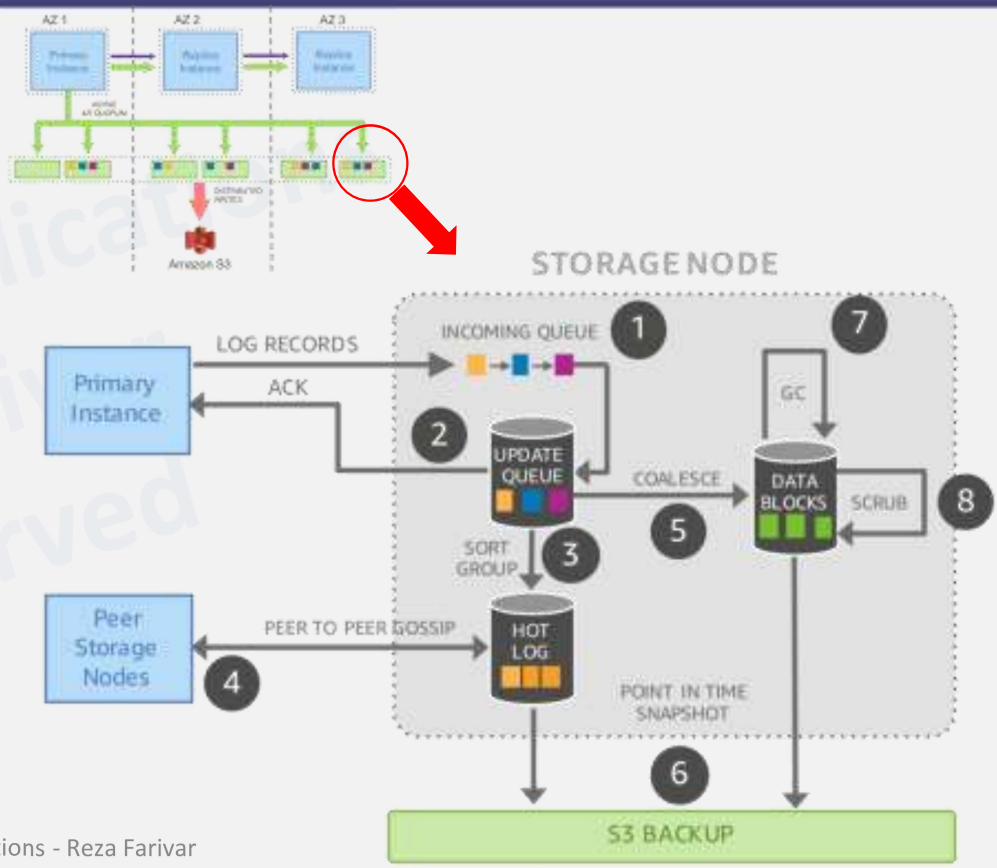
# Quorum Reads

- A quorum read is expensive, and is best avoided
- We do not need to perform a quorum read on routine page reads
  - It always knows where to obtain an up-to-date copy of a page
  - The client-side Aurora storage driver tracks which writes were successful for which segments
  - The driver tracks read latencies, and always tries to read from the storage node that has demonstrated the lowest latency in the past
- The only scenario when a quorum read is needed is during recovery on a database instance restart

# Amazon Aurora Storage Nodes

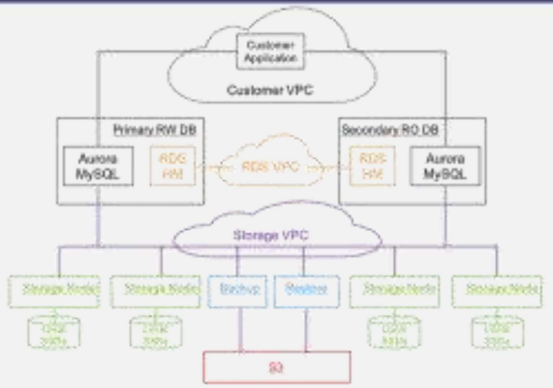
1. Receive log records and add to in-memory queue
2. persist record on disk and acknowledge, ACK to the database
3. Organize records and identify gaps in log since some batches may be lost
4. Gossip with peers to fill in holes
5. Coalesce log records into new page versions
6. Periodically stage log and new page versions to S3
7. Periodically garbage-collect old versions
8. Periodically validate CRC codes on blocks

- Each of the steps above are asynchronous
- Only steps (1) and (2) are in the foreground path potentially impacting latency



# Database Engine Implementation

- The database engine is a fork of “community” MySQL/InnoDB and diverges primarily in how InnoDB reads and writes data to disk
  - In community InnoDB, a write operation results in data being modified in buffer pages, and the associated redo log records written to buffers of the WAL in LSN order
  - On transaction commit, the WAL protocol requires only that the redo log records of the transaction are durably written to disk
  - The actual modified buffer pages are also written to disk eventually through a double-write technique to avoid partial page writes
  - These page writes take place in the background, or during eviction from the cache, or while taking a checkpoint
- In addition to the IO Subsystem, InnoDB also includes the transaction subsystem, the lock manager, a B+-Tree implementation and the associated notion of a “mini transaction” (MTR).
  - An MTR is a construct only used inside InnoDB and models groups of operations that must be executed atomically (e.g., split/merge of B+-Tree pages).
- Concurrency control is implemented entirely in the database engine without impacting the storage service

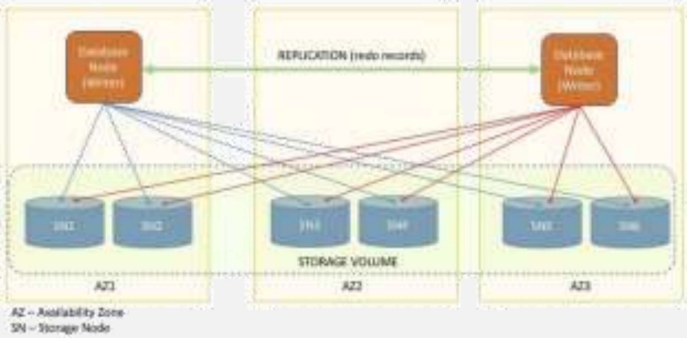


# Aurora and Consensus

- Aurora leverages only quorum I/Os, locally observable state, and monotonically increasing log ordering to provide high performance, non-blocking, fault-tolerant I/O, commits, and membership changes
- Aurora is able to avoid much of the work of consensus by recognizing that, during normal forward processing of a system, there are local oases of consistency
- Using backward chaining of redo records, a storage node can tell if it is missing data and gossip with its peers to fill in gaps
- Using the advancement of segment chains, a database instance can determine whether it can advance durable points and reply to clients requesting commits
- The use of monotonically increasing consistency points – SCLs, PGCLs, PGMRPLs, VCLs, and VDLs – ensures the representation of consistency points is compact and comparable
  - These may seem like complex concepts but are just the extension of familiar database notions of LSNs and SCNs.
- The key invariant is that the log only ever marches forward.

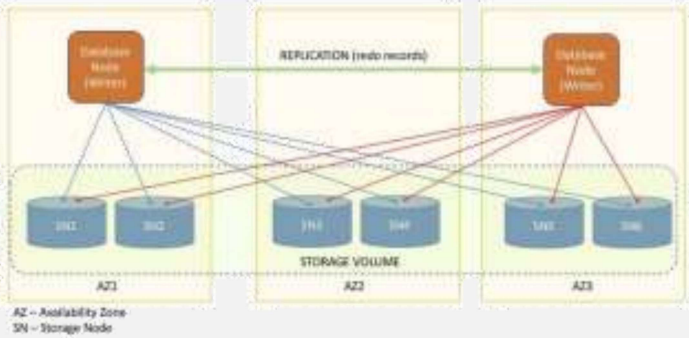
# Aurora Multi-Master

- For high availability and ACID transactions across a cluster of database nodes with configurable read after write consistency
- With single-master Aurora, a failure of the single writer node requires the promotion of a read replica to be the new writer
- In the case of Aurora Multi-Master, the failure of a writer node merely requires the application using the writer to open connections to another writer
- When designing for high availability, make sure that you are not overloading writers
- Conflicts arise when concurrent transactions or writes executing on different writer nodes attempt to modify the same set of pages



# Aurora Multi-Master Replication and Quorom

1. The application layer starts a write transaction
2. For cross-cluster consistency: The writer node proposes the change to all six storage nodes
3. Each storage node checks if the proposed change conflicts with a change in flight or a previously committed change and either confirms the change or rejects it
   - Each storage node compares the LSN (think of this as a page version) of the page submitted by the writer node with the LSN of the page on the node
   - It approves the change if they are the same and rejects the change with a conflict if the storage node contains a more recent version of the page
4. If the writer node that proposed the change receives a positive confirmation from a quorum of storage nodes:
   1. First, it commits the change in the storage layer, causing each storage node to commit the change
   2. It then replicates the change records to every other writer node in the cluster using a low latency, peer-to-peer replication protocol
5. The peer writer nodes, upon receiving the change, apply the change to their in-memory cache (buffer pool)
6. If the writer node that proposed the change does not receive a positive confirmation from a quorum of storage nodes, it cancels the entire transaction and raises an error to the application layer (The application can then retry the transaction)
7. Upon successfully committing changes to the storage layer, writer nodes replicate the redo change records to peer writer nodes for buffer pool refresh in the peer node



# Aurora vs. RDS

- RDS offers a greater range of database engines and versions than Aurora RDS
- Aurora RDS offers superior performance to RDS due to the unique storage subsystem
- Aurora RDS offers superior scalability to RDS due to the unique storage subsystem
- The pricing models differ slightly between RDS and Aurora RDS, but Aurora RDS is generally a bit more expensive to implement for the same database workload
- Aurora RDS offers superior high availability to RDS due to the unique storage subsystem
- According to AWS, Aurora offers five times the throughput of standard MySQL, performance on-par with commercial databases, but at one-tenth the cost
  - It should be noted that these numbers are AWS marketing claims.
  - House of Brick has found the cost claims to be roughly correct when compared with Oracle Enterprise Edition deployments, but the performance advantage of Aurora in real-world scenarios is closer to 30%

    *http://houseofbrick.com/aws-rds-mysql-vs-aurora-mysql/*

CLOUD COMPUTING APPLICATIONS

# Cloud Databases – Google Cloud Spanner

Prof. Reza Farivar

# Google Cloud Spanner

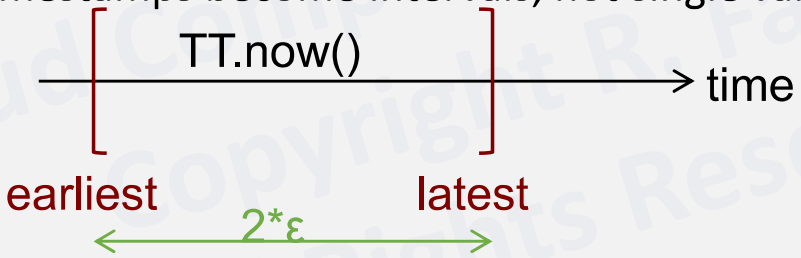
- Spanner is a distributed data layer that uses optimized sharded Paxos to guarantee consistency even in a system that spans multiple geographic regions
  - Query API is SQL (i.e. SELECT)
  - Insert and updates are done through a specialized GRPC interface
- Two-phase commit to achieve serializability
- TrueTime for external consistency, consistent reads without locking, and consistent snapshots
  - *External > Strong > Weak*

# Spanner and CAP

- Is Spanner C+A+P?
  - No, It is CP
  - during (some) partitions, Spanner chooses C and forfeits A
- Availability is in the 5 nines range
  - Is this acceptable to your application?
  - Effectively CA
- Spanner uses the Paxos algorithm as part of its operation to shard (partition) data across hundreds of servers
- 2PC known as the anti-availability protocol
  - because all members must be up for it to work
  - In Spanner, each member is a Paxos group
  - ensures each 2PC “member” is highly available even if some of its Paxos participants are down
- Cloud Spanner provides stale reads, which offer similar performance benefits as eventual consistency but with much stronger consistency guarantees
  - A stale read returns data from an "old" timestamp, which cannot block writes because old versions of data are immutable

# TrueTime

- Heavy use of hardware-assisted clock synchronization using GPS clocks and atomic clocks to ensure global consistency
  - avoid communication in a distributed system
  - GPS and Atomic clock have different failure modes
- "Global wall-clock time" with bounded uncertainty
  - ε is worst-case clock divergence
  - Timestamps become intervals, not single values




| Method | Returns |
|---|---|
| $TT.now()$ | $TTinterval$: $[earliest, latest]$ |
| $TT.after(t)$ | true if $t$ has definitely passed |
| $TT.before(t)$ | true if $t$ has definitely not arrived |

- Consider event $e_{now}$ which invoked tt = TT.now():
  - Guarantee: $tt.earliest <= t_{abs}(e_{now}) <= tt.latest$

# Spanner

- TrueTime exposes clock uncertainty
  - Commit wait ensures transactions end after their commit time
  - Read at TT.now.latest()
- Reads dominant, make them lock-free
  - Read-Only Transaction
    - A replica can satisfy a read at a timestamp t if t <= $t_{safe}$.
  - Snapshot Read, client-provided timestamp
    - read at a particular time in the past
  - Snapshot Read, client-provided bound
- Read-Write Transaction less common
  - Pessimistic, use 2 phase locking
- Globally-distributed database
  - 2PL w/ 2PC over Paxos!

$$t_{safe} = min(t_{safe}^{Paxos}, t_{safe}^{TM})$$

Paxos State Machine Safe Time

Transaction Manager Safe Time

# HW-based Time Synchronization

- NTP
  - Software time synchronization with one server
  - Stratum 2
  - Network delays
  - 1/2 to 100 ms accuracy
- Google Spanner
- Microsoft Azure now offers GPS clock synchronization
  - VMICTimeSync provider
  - Precision Time Protocol
  - Stratum 1 devices
    - I.e. direct connection, not through shared network, to a reference time server (Stratum 0)
    - 10 microseconds accuracy
      - For reference, GPS accuracy < 1 us, 95% of the time ≤40 nanoseconds
- Amazon Time Sync Service
  - Chrony vs. NTP

CLOUD COMPUTING APPLICATIONS

# Cloud Databases – Microsoft Azure CosmosDB

Prof. Reza Farivar

# Azure CosmosDB

- Globally distributed, multi-model database
  - Wire compatible with Cassandra
  - Table API
    - 5 types of consistency levels
  - MongoDB API
  - Etcd API
    - etcd is a consistent distributed key value storage
    - Compare with Zookeeper
    - Backend of Kubernetes
  - Gremlin API
- Replicated State Machine (RSM) for concurrency
  - Specific algorithm not published
  - *RAFT is also a state machine consensus algorithm*

# Azure CosmosDB

- Write-optimized, resource-governed and schema-agnostic database engine
  - It automatically indexes everything it ingests
    - Internally based on a document store model
    - All JSON documents are a tree
    - Make an index for each path of the tree
  - **Synchronously** makes the index durable and highly available before acknowledging the client's updates while maintaining low latency guarantees
- BW-Tree indexing
  - Log Structure Record
  - Latch free updates
  - Atom-record-sequence (ARS) system
- Uses Lamport's TLA+ specification language to describe SLAs at each consistency level

# CosmosDB Consistency Models

- **Strong:** With strong consistency, you are guaranteed to always read the latest version of an item similar to read committed isolation in SQL Server. You can only ever see data which is durably committed. Strong consistency is scoped to a single region.
- **Bounded-staleness:** In bounded-staleness consistency read will lag behind writes and guarantees global order and not scoped to a single region. When configuring bounded-staleness consistency you need to specify the maximum lag by:
  - Operations: For a single region the maximum operations lag must be between 10 and 1,000,000, and for the multi region, it will be between 100,000 and 1,000,000.
  - Time: The maximum lag must be between 5 seconds and 1 day for either single or multi-regions.
- **Session:** This is the most popular consistency level, since it provides consistency guarantees but also has better throughput.
- **Consistent Prefix:** A global order is preserved, and prefix order is guaranteed. A user will never see writes in a different order than that is which it was written.
- **Eventual:** Basically, this is like asynchronous synchronization. It guarantees that all changes will be replicated eventually, and as such, it also has the lowest latency because it does not need to wait on any commits.


**Figure 4. Multiple well-defined consistency choices along the spectrum.**

| Consistency Level | Guarantees |
|---|---|
| Strong | Linearizability |
| Bounded Staleness | Consistent Prefix. Reads lag behind writes by *k* prefixes or *t* interval |
| Session | Consistent Prefix. Monotonic reads, monotonic writes, read-your-writes, write-follows-reads |
| Consistent Prefix | Updates returned are some prefix of all the updates, with no gaps |
| Eventual | Out of order reads |

# Azure CosmosDB Implementation

- Cosmos DB service is deployed on several replicated shared-nothing nodes across geographical regions for high-availability, low-latency, and high throughput.
- Some or all of these distributed nodes form a replica set for serving requests on a data shard that contains documents.
- Among the replicas, one of them is elected as a master to perform totally-ordered writes on the data shard.
- Writes are done on the write-quorum (W), a subset of the replica nodes, to ensure that the data is durable.
- Reads are performed on read-quorum (R), a subset of replica nodes, to get the desired consistency levels (Strong, Bounded-staleness, Session, Consistent Prefix, Eventual) as configured by users.
- Data is partitioned at logic level and is replicated at storage layer in terms of physical partitions to achieve desired availability and throughput.
- storage
  - transactional storage engine
  - analytical storage engine
  - storage engines are log-structured and write-optimized

CLOUD COMPUTING APPLICATIONS

# Cloud Databases - NoSQL

Prof. Reza Farivar

# Key/value Databases

- Key-value databases are optimized for common access patterns, typically to store and retrieve large volumes of data
- These databases deliver quick response times, even in extreme volumes of concurrent requests

- High-traffic web apps, ecommerce systems, and gaming apps
- AWS DynamoDb
- Azure CosmosDB

# Wide Column Databases

- Google BigTable
  - Cloud Bigtable is a fully managed, wide-column NoSQL database that offers low latency and replication for high availability
  - This is what HBase was modeled after
- Managed Casandra
  - *Cassandra was modeled after Dynamo (paper)*
  - *DynamoDB was modeled after Casandra*
  - AWS managed Casandra
- Cassandra AMI for any cloud provider

# In Memory (Cache) Databases

- In-memory databases are used for applications that require real-time access to data
- By storing data directly in memory, these databases deliver microsecond latency to applications for whom millisecond latency is not enough
- Caching, gaming leaderboards, and real-time analytics
- Common usage pattern: Cache RDS or document databases

- AWS ElastiCashe (Redis / MemCached)
- Azure Cache for Redis
- Google Memorystore (Redis / MemCached)
- IBM Redis

# Document Databases

- A document database is designed to store semistructured data as JSON-like documents
- Makes it easy to store, ***query***, and index JSON data
- Content management, catalogs, and user profiles
- Non-relational database service

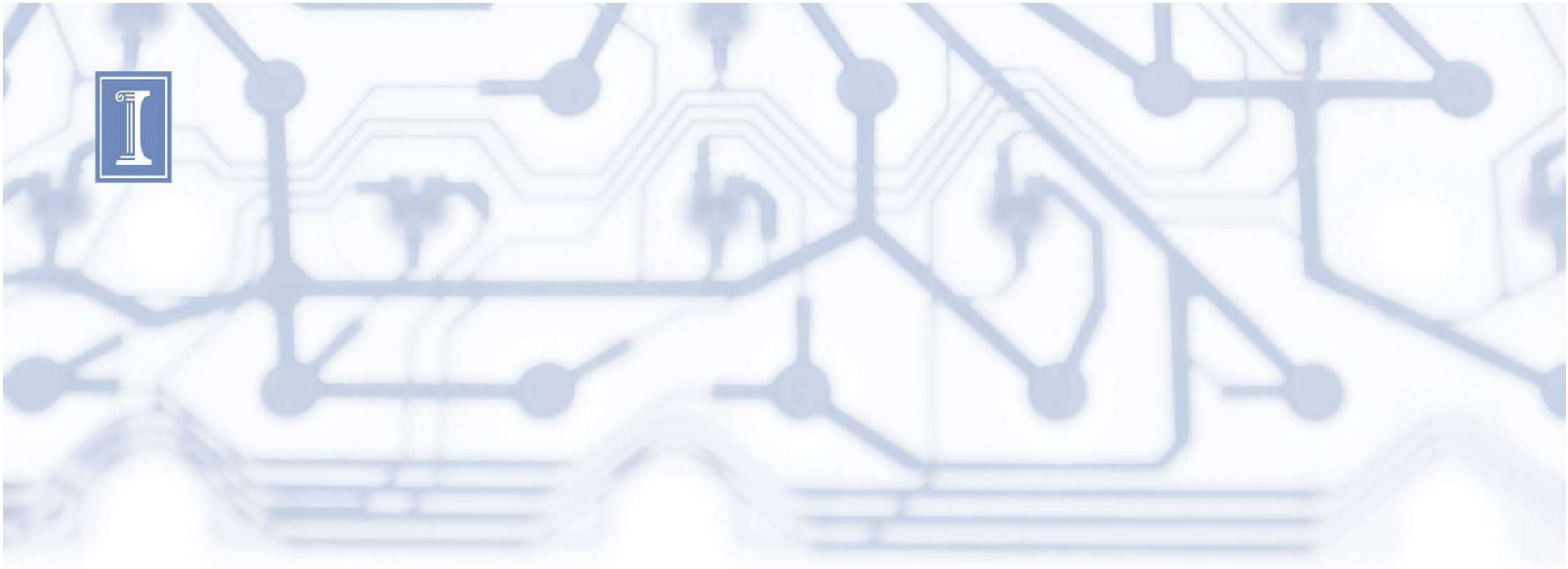
- AWS DocumentDB + (MongoDB compatibility)
- Azure CosmosDB
- Google Firestore
  - Targeted for mobile App support
- IBM Cloudant / IBM MongoDB

# AWS DocumentDB

- Managed instance
- Implements MongoDB 3.6 API
- Storage and compute are decoupled, allowing each to scale independently
- Automatically grows the size of storage volume as the database storage needs grow
  - Grows in increments of 10 GB, up to a maximum of 64 TB
- Up to 15 low latency read replicas to increase read capacity
- Replicates six copies of data across three AWS Availability Zones (AZs)
- Access to Amazon DocumentDB clusters must be done through the mongo shell or with MongoDB drivers

# Other Types of Cloud Databases

- Graph Databases
  - Covered in a different module
- Time Series Databases
  - AWS Timestream
- Blockchain / Ledgers
  - Immutable and cryptographically verifiable transactions
  - AWS QLDB
- Data Warehouses
  - Covered in a different module
  - Columnar storage
    - AWS Redshift
    - Google BigQuery
    - Azure Synapse (formerly Azure SQL Data Warehouse)



CLOUD COMPUTING APPLICATIONS

# Caching as a Universal Concept: Overview

Prof. Reza Farivar

# The need for caching

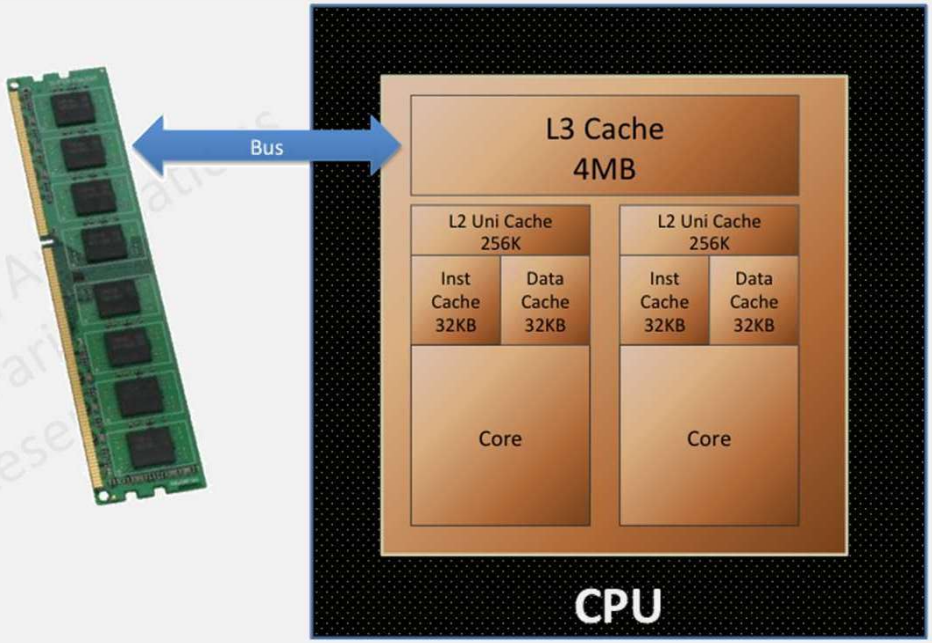
- Success for many websites and web applications relies on speed
  - Users can register a 250-millisecond (1/4 second) difference between competing sites
    - "***For Impatient Web Users, an Eye Blink Is Just Too Long to Wait***" NYT, 2012
  - For every 100-ms (1/10 second) increase in load time, sales decrease 1 percent
  - Data that is cached can be delivered much faster
- In-memory Key-value stores can provide sub-millisecond latency
  - querying a database is always slower and more expensive than locating a key in a key-value pair cache

# Caching

- Caching is a universal concept
- Based on the principle of locality (aka. locality of reference)
  - Tendency of the “processor” to access the same set of memory locations repetitively over a short period of time
  - Temporal locality vs spatial locality
- Whenever you have “large + slow” source of information and “small + fast” storage technology, you can use the latter to cache the former
- You can see this concept anywhere from CPUs and processors, to operating systems, to large web applications on the cloud

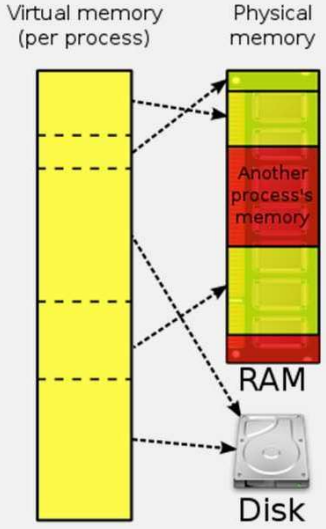
# Caching in Processors: Data & Instructions

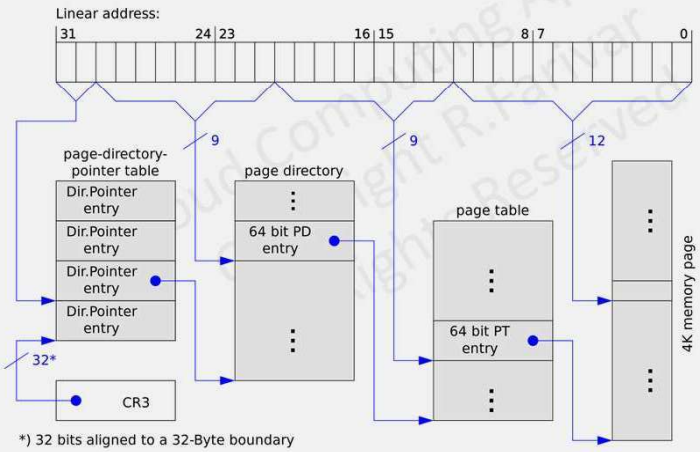
- Big/Slow RAM memory
- Multiple layers of caching
- Access to data exhibits temporal and spatial locality
  - L1 Data Cache, L2 and L3 Caches
- The program instructions have spatial and temporal locality
    - One instruction after another
    - Loops
  - Also branch locality
    - If … else

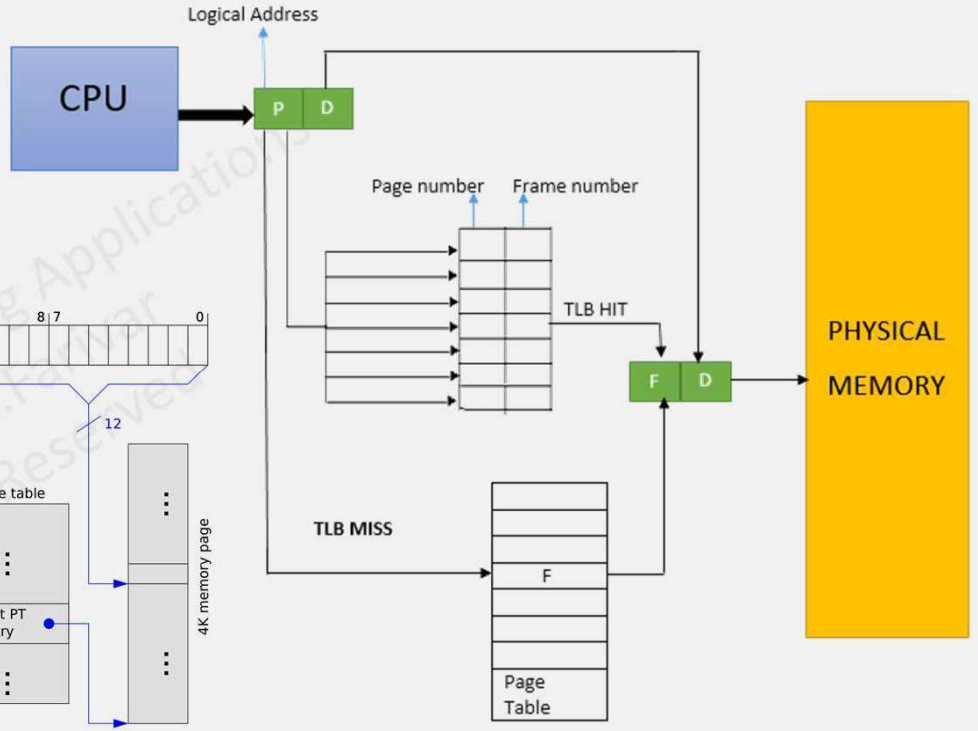


# Caching in Processors: Virtual Memory

- Virtual memory address translation
- Every memory access needs a translation
  - Needs a page walk
  - Slow/big source of data
- Translation Lookaside Buffer (TLB)
  - iTLB and dTLB



# Virtual memory and OS-level Page Caching

- Virtual memory:
  - Each process thinks it has $2^{48}$ = 256 TB of memory
- Paging
  - computer stores and retrieves data from secondary storage (HDD/SSD) for use in main memory
  - RAM acts as the “cache” for the SSD
  - When a process tries to reference a page not currently present in RAM, the processor treats this invalid memory reference as a ***page fault*** and transfers control from the program to the operating system
- OS does the following
  - Determine the location of the data on disk
  - Obtain an empty page frame in RAM to use as a container for the data
  - Load the requested data into the available page frame
  - Update the page table to refer to the new page frame
  - Return control to the program, transparently retrying the instruction that caused the page fault

# Linux Page Cache

- Linux kernels up to version 2.2 had both a Page Cache as well as a Buffer Cache. As of the 2.4 kernel, these two caches have been combined. Today, there is only one cache, the Page Cache.
- This mechanism also caches files.
- Usually, all physical memory not directly allocated to applications is used by the operating system for the page cache
- So the OS keeps other pages that it may think may be needed in the page cache.
- If Linux needs more memory for normal applications than is currently available, areas of the Page Cache that are no longer in use will be automatically deleted.

```
wfischer@pc:~$ dd if=/dev/zero of=testfile.txt bs=1M count=10
10+0 records in
10+0 records out
10485760 bytes (10 MB) copied, 0,0121043 s, 866 MB/s
wfischer@pc:~$ cat /proc/meminfo | grep Dirty
Dirty:             10260 kB
wfischer@pc:~$ sync
wfischer@pc:~$ cat /proc/meminfo | grep Dirty
Dirty:                 0 kB
```

*https://www.thomas-krenn.com/en/wiki/Linux_Page_Cache_Basics*

# Linux VFS Cache

- Dentry Cahce
  - A "dentry" in the Linux kernel is the in-memory representation of a directory entry
  - A way of remembering the resolution of a given file or directory name without having to search through the filesystem to find it
  - The dentry cache speeds lookups considerably; keeping dentries for frequently accessed names like `/tmp`, `/dev/null`, or `/usr/bin/tetris` saves a lot of filesystem I/O.
- Inode Cache
  - As the mounted file systems are navigated, their VFS inodes are being continually read and, in some cases, written
  - he Virtual File System maintains an inode cache to speed up accesses to all of the mounted file systems
  - Every time a VFS inode is read from the inode cache the system saves an access to a physical device.

# Caching in Distributed Systems

- CDN Caching
- Web Server Caching
  - Reverse Proxies
  - Varnish
  - Web servers can also cache requests, returning responses without having to contact application servers
    - NGINX
- Database Caching
- Application Caching
  - In-memory caches such as Memcached and Redis are key-value stores between your application and your data storage

# What to Cache

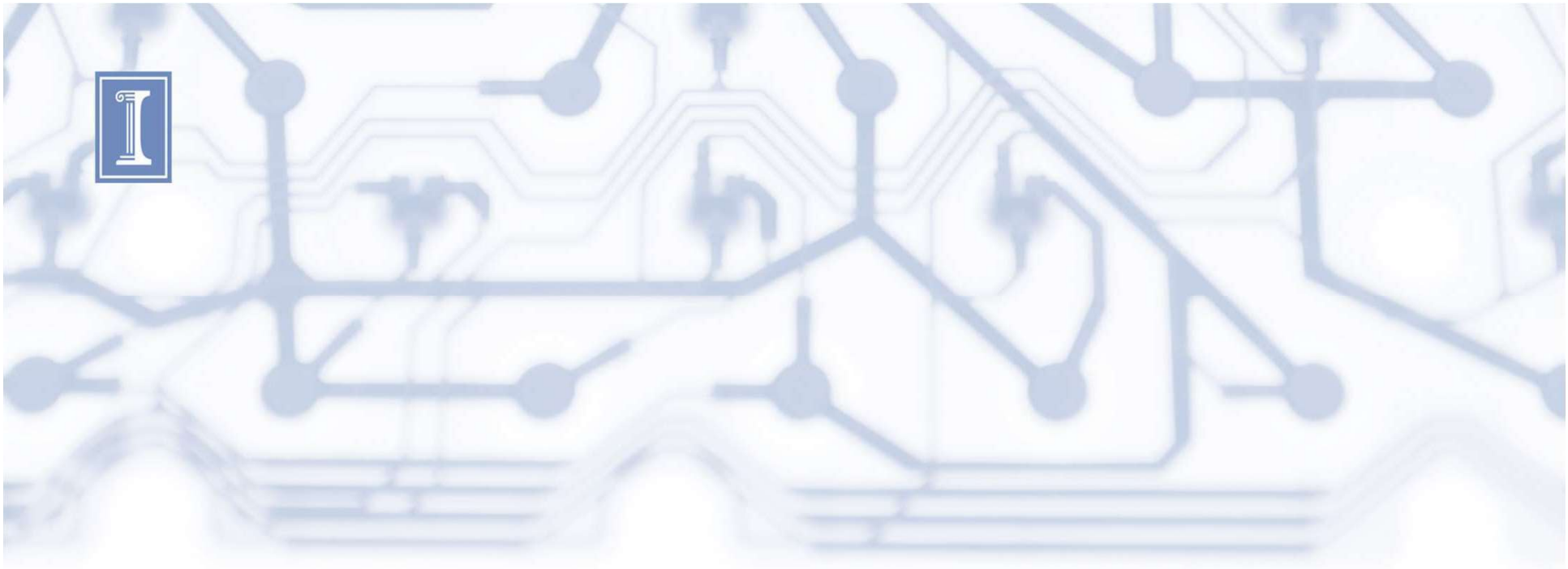
- There are multiple levels you can cache that fall into two general categories: **database queries** and **objects**:
  - Row level
  - Query-level
  - Fully-formed serializable objects
  - Fully-rendered HTML

# Caching at the database query level

- Whenever you query the database, hash the query as a key and store the result to the cache
- Suffers from expiration issues:
  - Hard to delete a cached result with complex queries
  - If one piece of data changes such as a table cell, you need to delete all cached queries that might include the changed cell

# Caching at the object level

- See your data as an object, similar to what you do with your application code. Have your application assemble the dataset from the database into a class instance or a data structure(s):
  - Remove the object from cache if its underlying data has changed
  - Allows for asynchronous processing: workers assemble objects by consuming the latest cached object
- Suggestions of what to cache:
  - User sessions
  - Fully rendered web pages
  - Activity streams
  - User graph data



CLOUD COMPUTING APPLICATIONS

# Caching Technical Concepts

Prof. Reza Farivar

# Topics

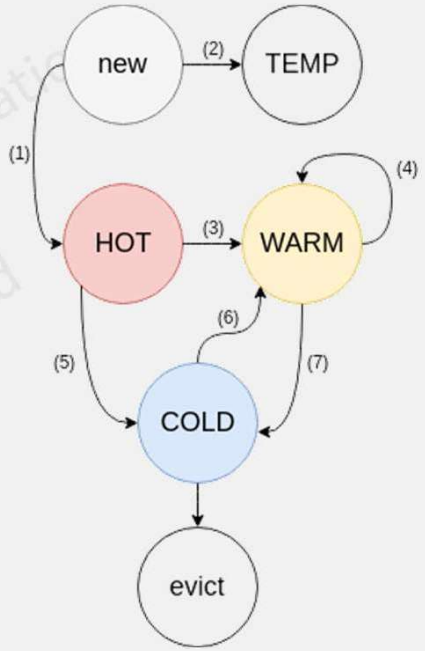
- Cache Replacement Policy
  - LRU, FIFO, etc.
- Cache Writing / Updating Policy
  - Cache Aside, Write-through, Write-back
- Cache Coherence

# Reading from Cache

- A client first checks to see if a data piece is available in a cache
- Cache hit: the desired data is in the cache
- Cache miss: the desired data is not in the cache
  - In this case, the client needs to go to the "slow/big" source of data
  - Once data is retrieved, it is also copied in the cache, ready for next accesses (principle of locality)
    - But **where** in the cache should we put this new data?

# Cache Replacement Policy

- Also known as:
  - Cache Eviction Policy
  - Cache Invalidation Algorithm
- Least Recently Used (LRU)
  - Replaces the oldest entry in the cache
  - Memcached uses segmented LRU
- Time-aware Least Recently Used (TLRU)
  - TTU: Time To Use
- Least Frequently Used (LFU)
- First in First Out (FIFO)
- Many others
  - LIFO, FILO, MRU, PLRU,



# Cache Replacement

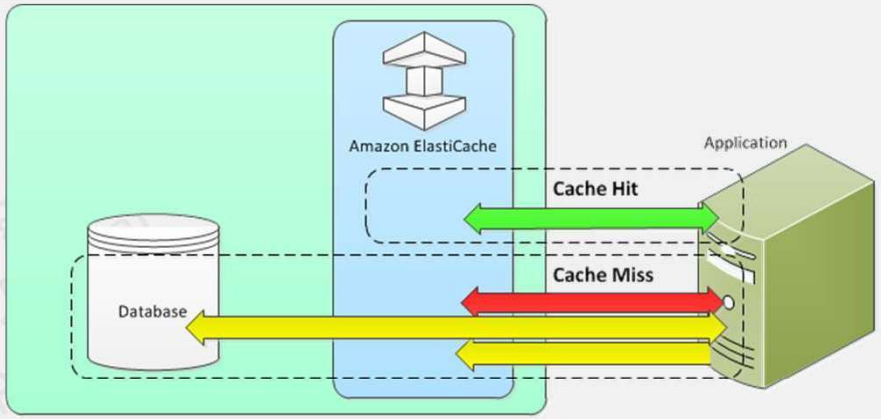
- maintain consistency between caches and the source of truth such as the database through cache invalidation
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Time to Live (TTL): Expiration time for records in the cache
  - Eventual consistency for coherence problem

# Cache Writing/Updating Policies

- Whenever a new piece of data is created and needs to be stored, we also have a question regarding updating the caches
  - Cache Aside
    - Lazy Loading
  - Write-Through
    - Write is done synchronously both to the cache and to the backing store.
  - Write – Back
    - Write-Behind
    - Lazy Writing

# Cache Aside (aka lazy loading)

- The ***application*** is responsible for reading and writing from storage
  - The cache does not interact with storage directly
- The ***application*** does the following:
  - Look for entry in cache, resulting in a cache miss
  - Load entry from the database
  - Add entry to cache
  - Return entry
- Memcached is usually used in this manner



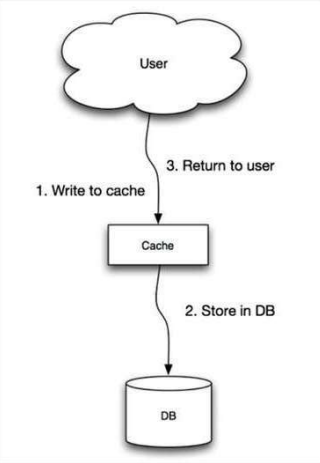

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return user
```

# Disadvantages of Cache-aside

- Each cache miss results in three trips, which can cause a noticeable delay.
- Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
- When a node fails, it is replaced by a new, empty node, increasing latency.

# Write-through writing/updating policy

- The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:
  - Application adds/updates entry in cache
  - Cache synchronously writes entry to data store
  - Return




Application code:

```
set_user(12345, {"foo":"bar"})
```

Cache code:

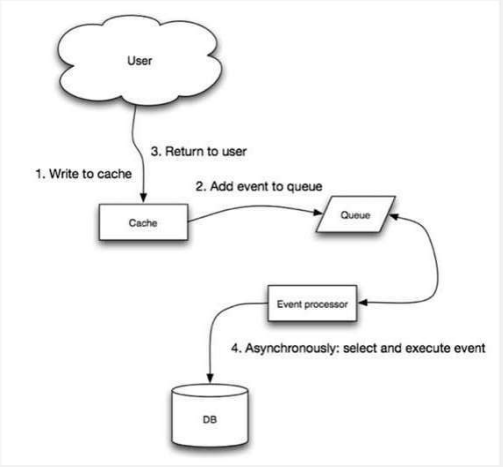
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

# Disadvantages of Write-Through

- Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast
  - Users are generally more tolerant of latency when updating data than reading data.
- Data in the cache is never stale ☺
- When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database
  - Cache-aside in conjunction with write through can mitigate this issue
- ***Cache Churn:*** Most data written might never be read → waste of resource
  - Can be minimized with a TTL

# Write-back (Write-Behind) writing/updating policy

- Initially, writing is done only to the cache.
- The write to the backing store is postponed until the modified content is about to be replaced by another cache block.
  - Asynchronously write entry to the data store, improving write performance
- Write-back cache is more complex to implement, since it needs to track which of its locations have been written over, and mark them as *dirty* for later writing to the backing store
  - ***Lazy Write***: Data in these dirty locations are written back to the backing store only when they are evicted from the cache
- a read miss in a write-back cache (which requires a block to be replaced by another) will often require two memory accesses to service: one to write the replaced data from the cache back to the store, and then one to retrieve the needed data.




*Img source:* Scalability, Availability & Stability Patterns
*Jonas Bonér*

# Write-around

- Writing is only done to the underlying data source
- **Advantage:** Good for not flooding the cache with data that may not subsequently be re-read
- **Disadvsntage:** Reading recently written data will result in a cache miss (and so a higher latency) because the data can only be read from the slower backing store
- The write-around policy is good for applications that don't frequently re-read recently written data
  - This will result in lower write latency but higher read latency which is an acceptable trade-off for these scenarios

# Write Allocation Policies

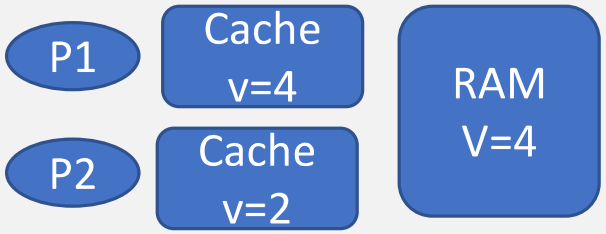
- Since no data is returned to the requester on write operations, a decision needs to be made on write misses, whether or not data would be loaded into the cache. This is defined by these two approaches:
  - *Write allocate* (also called *fetch on write*): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
  - *No-write allocate* (also called *write-no-allocate* or *write around*): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.

# Write Allocation Policies

- A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.
- A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, since they still need to be written directly to the backing store.

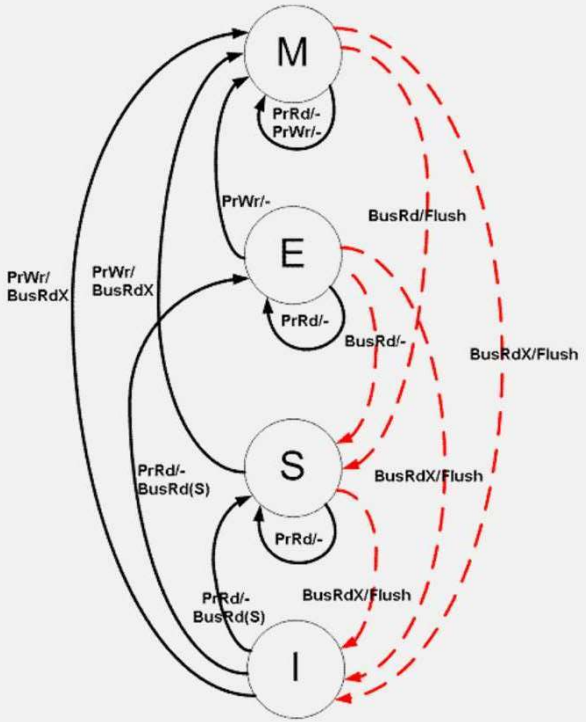
# Cache Coherency

- *Note: do not confuse with Consistency!*
- Whenever cache is distributed, ***with the same data in more than one place***, we have the coherency problem
  - Write Propagation: Changes to the data in any cache must be propagated to other copies (of that value) in the peer caches
  - Transaction Serialization: Reads/Writes to a single memory location must be seen by all processors in the same order
- Coherence Protocols
  - **Write-invalidate**: When a write operation is observed to a location that a cache has a copy of, the cache controller ***invalidates*** its own copy
    - **MESI** protocol
  - **Write-Update**: When a write operation is observed to a location that a cache has a copy of, the cache controller ***updates*** its own copy



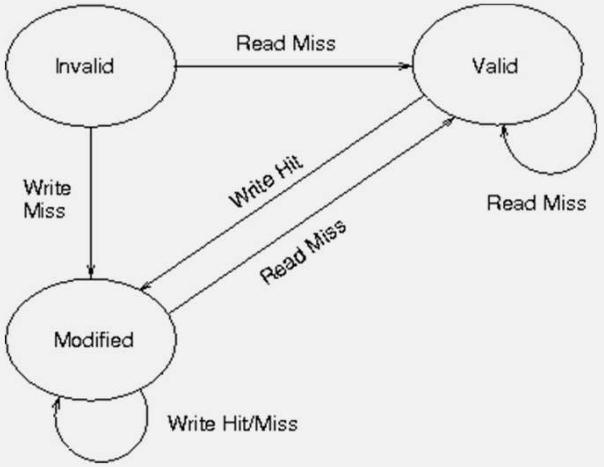
# MESI protocol

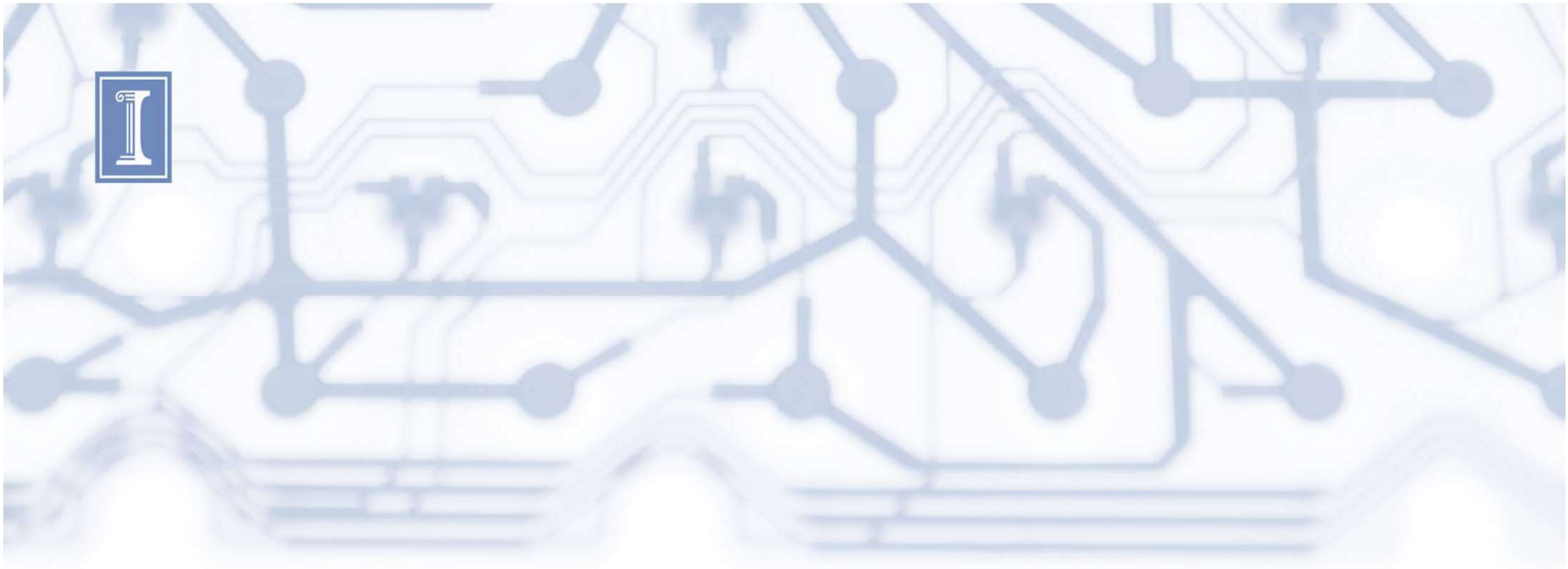
- Aka the Illinois Protocol
- Supports “write-back-caches”
- Uses Cache-to-cache transfer
- States:
  - Modified
  - Exclusive
  - Shared
  - Invalid



# Coherency Mechanisms

- Snooping
  - Send all requests for data to all processors
  - Processors “snoop” to see if they have a local copy
  - Requires broadcast
  - Works well with a “bus” → CPUs
  - Usually implemented with state machines
- Directory-based
  - Keep track of what is being share in a centralized location
    - In reality, every block in every cache
  - Send point-to-point requests
  - Scales better than snooping
  - Lcache, WP-Lcache
- TTL: Eventually consistent caches
  - Main mechanism for DNS caching records





CLOUD COMPUTING APPLICATIONS

# AWS ElastiCache for Memcached

Prof. Reza Farivar

# MemCached

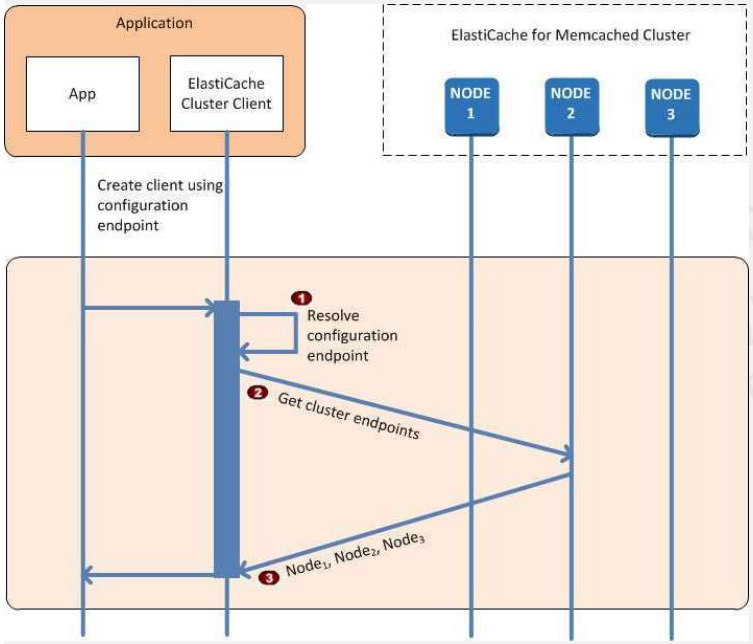
- Simple key/value storage model
  - E.g. in a Lambda Python function, just import pymemcache and set / get data
  - Other libraries: python-Memcached, pylibmc, twisted-Memcached, etc.
- Simple data model:
  - Strings, Objects
  - From the results of database calls, API calls, even HTML page renderings (e.g. PHP to HTML)
- No data storage
  - If a node goes down, you lose all the data in that node's memory
- Multi-threaded engine
- Multi-cluster using consistent hashing
  - Default limit 20 nodes
- Auto Discovery

# ElastiCache Memcached Auto Discovery

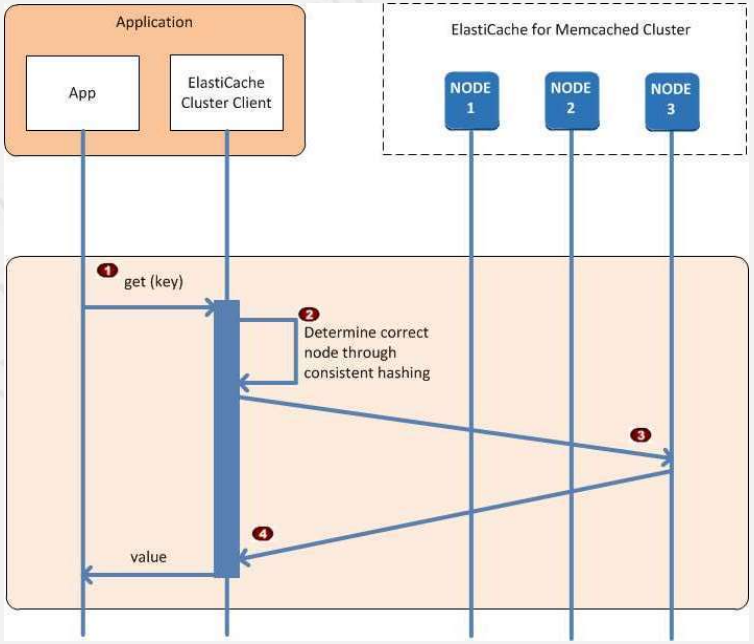
- Configuration endpoint
  - When you increase the number of nodes in a cache cluster, the new nodes register themselves with the configuration endpoint and with all of the other nodes
- When you remove nodes from the cache cluster, the departing nodes deregister themselves.
- In both cases, all the other nodes in the cluster are updated with the latest cache node metadata.
- Cache node failures are automatically detected; failed nodes are automatically replaced.
- A client program only needs to connect to the configuration endpoint.
- After that, the Auto Discovery library connects to all of the other nodes in the cluster.
- Client programs poll the cluster once per minute (this interval can be adjusted if necessary). If there are any changes to the cluster configuration, such as new or deleted nodes, the client receives an updated list of metadata. Then the client connects to, or disconnects from, these nodes as needed.

# ElastiCache for Memcached Auto Discovery

Connecting to Cache nodes




Normal operation




Cluster Client available for Java, .Net and PHP

# Example use of Elasticache for Memcached

```
from __future__ import print_function
import time
import uuid
import sys
import socket
import elasticache_auto_discovery
from pymemcache.client.hash import HashClient

#elasticache settings
elasticache_config_endpoint = "your-elasticache-cluster-endpoint:port"
nodes = elasticache_auto_discovery.discover(elasticache_config_endpoint)
nodes = map(lambda x: (x[1], int(x[2])), nodes)
memcache_client = HashClient(nodes)

def handler(event, context):
    """
    This function puts into memcache and get from it.
    Memcache is hosted using elasticache
    """

    #Create a random UUID... this will be the sample element we add to the cache.
    uuid_inserted = uuid.uuid4().hex
    #Put the UUID to the cache.
    memcache_client.set('uuid', uuid_inserted)
    #Get item (UUID) from the cache.
    uuid_obtained = memcache_client.get('uuid')
    if uuid_obtained.decode("utf-8") == uuid_inserted:
        # this print should go to the CloudWatch Logs and Lambda console.
        print ("Success: Fetched value %s from memcache" %(uuid_inserted))
    else:
        raise Exception("Value is not the same as we put :(. Expected %s got %s" %(uuid_inserted, uuid_obtained))

    return "Fetched value from memcache: " + uuid_obtained.decode("utf-8")
```

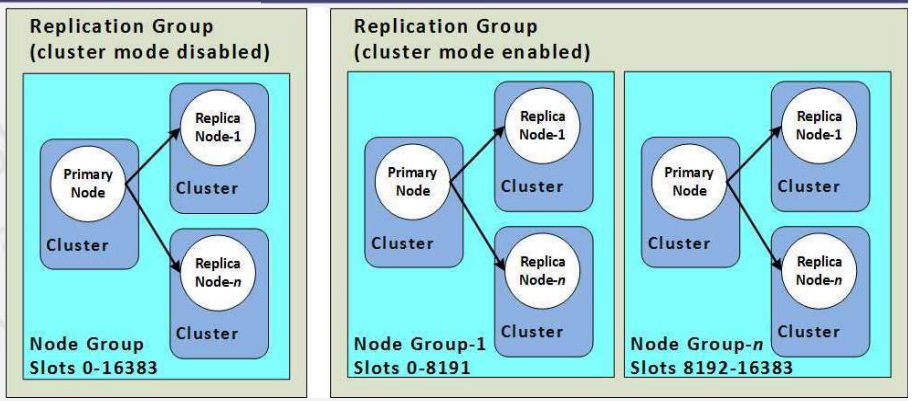
CLOUD COMPUTING APPLICATIONS

# AWS ElastiCache for Redis

Prof. Reza Farivar

# ElastiCache for Redis

- Up to 250 shards
- Each shard can be on a node-group
- Each node group can have one master (Write + read) and 5 other read replicas
  - If any primary has no replicas and the primary fails, you lose all that primary's data
- Node or shard limit of 500 in Redis 5.0.6+
  - 83 shards (one primary and 5 replicas per shard)
  - 500 shards (single primary and no replicas)

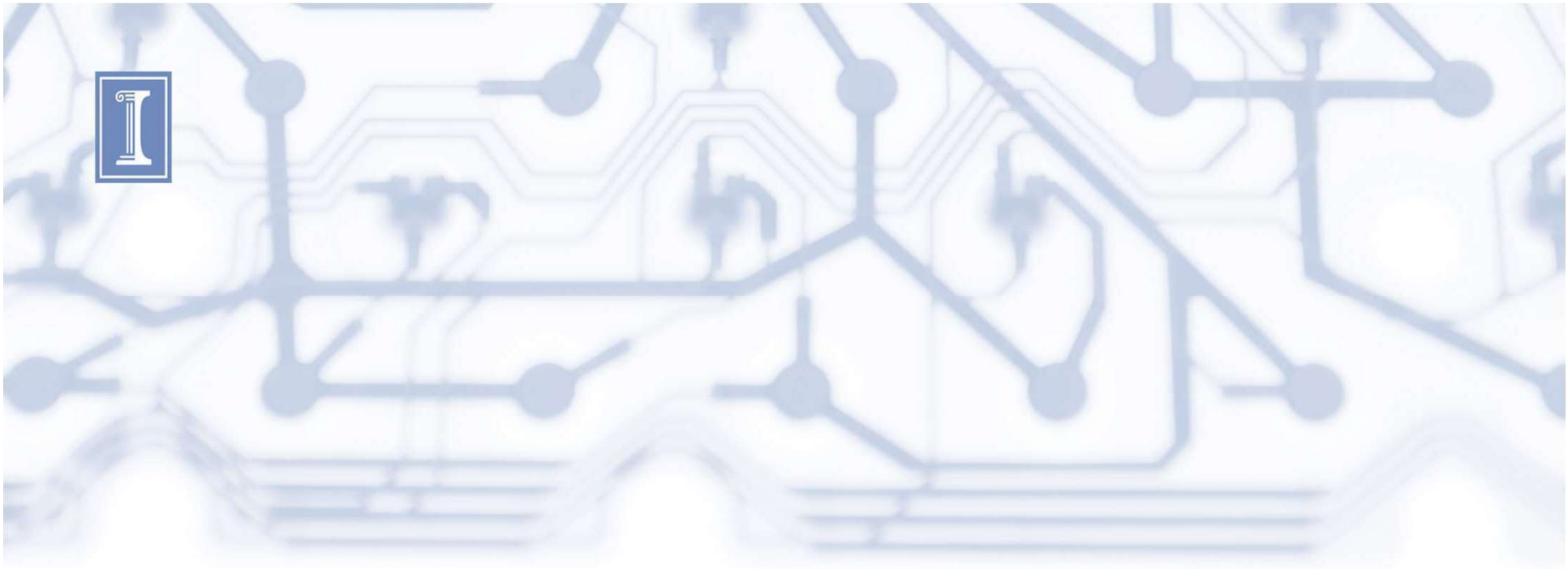


# Designing the Right Cache

- At the highest level, Memcached is generally used to store small and static data, such as HTML code pieces
  - Memory management is efficient and simple
  - No data persistence
  - If any node/cluster fails Memcached data is lost
    - Use Memcached with easily recoverable data
- Redis supports more complex data structures
  - Fast performance
  - Persistent storage
  - Read replicas

# Comparing Memcached and Redis

- Memcahced
  - Simple model
    - Strings, Objects
  - Large nodes, multithreading
  - Ability to scale out, and scale in
- Redis
  - Complex Data Types
    - Strings, Hashes, lists, sets, sorted sets, bitmaps
    - Sort in-memory datasets
  - Persistence of the key store
  - Replicate data for read-access to up to 5 read replicas per shard
    - Automatic failover if the primary node fail
  - Authenticate users with role-based access control
  - Redis streams: log data structure, producers append new data, consumers consume messages
  - Encryption
    - HIPAA eligible, PCI DSS, FedRAMP
  - Dynamically adding / removing shards from cluster mode Redis
    - Online resharding



CLOUD COMPUTING APPLICATIONS

# Cloud Caching Strategies

Prof. Reza Farivar

# Caching Strategies

- strategies to implement for populating and maintaining your cache depend upon what data you cache and the access patterns to that data
  - Cache Aside (Lazy Loading)
  - Write-Through
  - Adding TTL
- Deploying nodes to multiple Availability Zones (Elasticache supports this) can avoid single point of failure and provides high availability