CLOUD COMPUTING APPLICATIONS

Caching as a Universal Concept: Overview

Prof. Reza Farivar

The need for caching

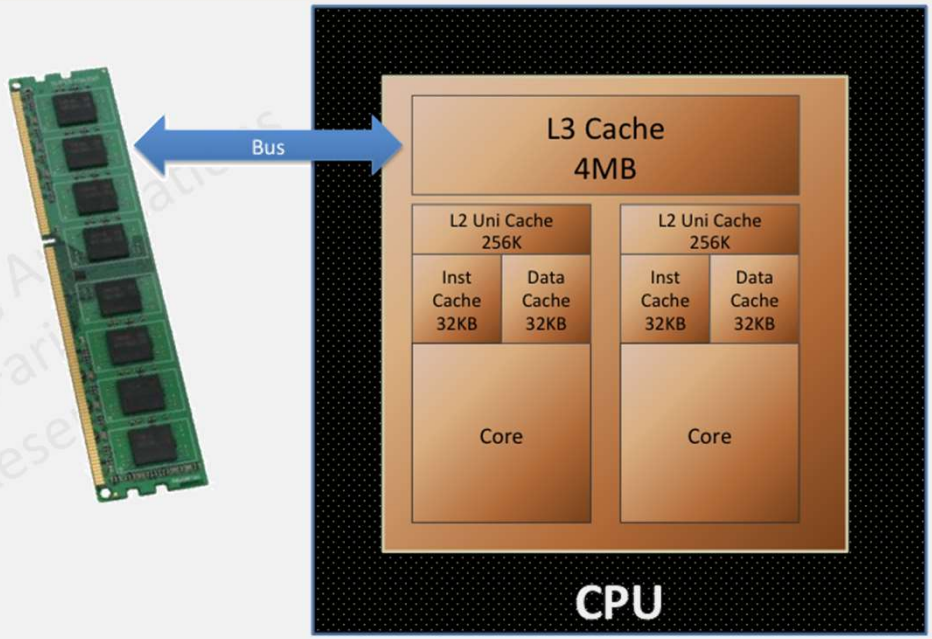
- Success for many websites and web applications relies on speed
  - Users can register a 250-millisecond (1/4 second) difference between competing sites
    - "***For Impatient Web Users, an Eye Blink Is Just Too Long to Wait***" NYT, 2012
  - For every 100-ms (1/10 second) increase in load time, sales 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

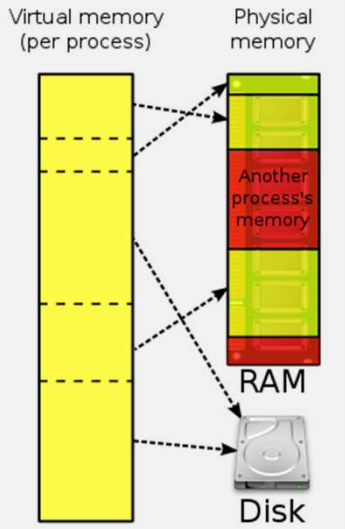
Caching in Processors: Data & Instructions

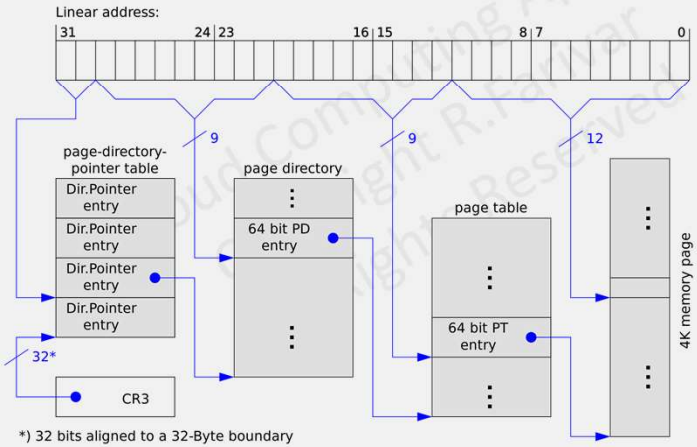
- Big/Slow RAM memory
- Multiple layers of caching
- Access to data exhibits temporal and spatial locality
  - L1 Data Cache, L2 and L3 Caches
- The program instructions have spatial and temporal locality
    - One instruction after another
    - Loops
  - Also branch locality
    - If … else

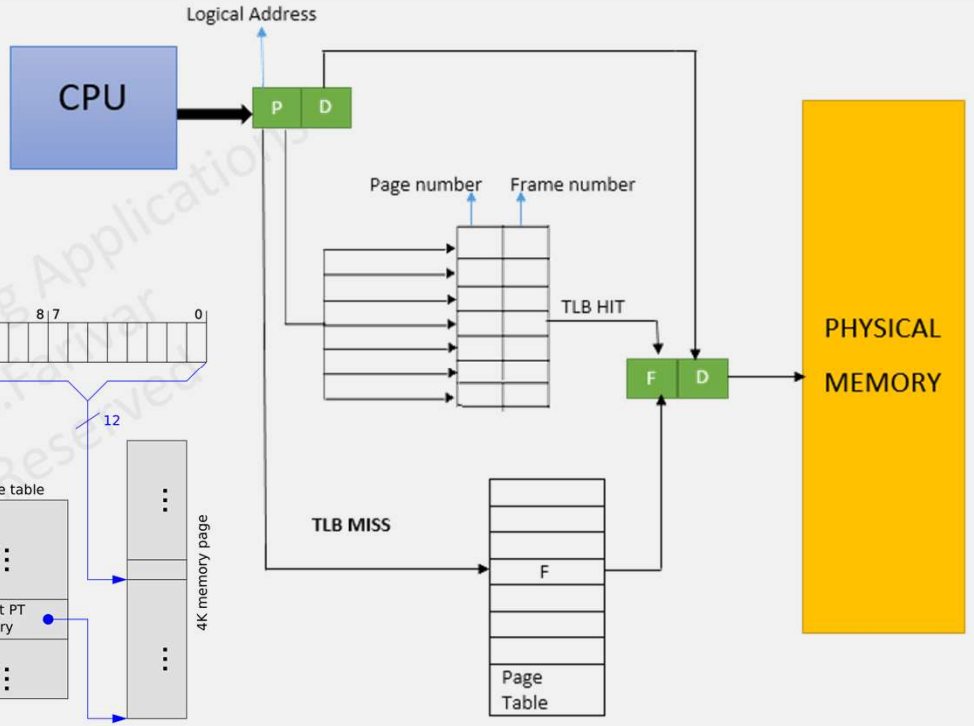


Caching in Processors: Virtual Memory

- 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

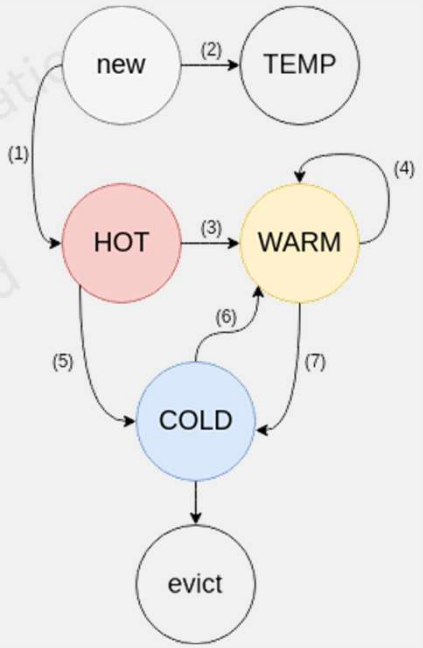
- Cache Replacement Policy
  - LRU, FIFO, etc.
- Cache Writing / Updating Policy
  - Cache Aside, Write-through, Write-back
- Cache Coherence

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

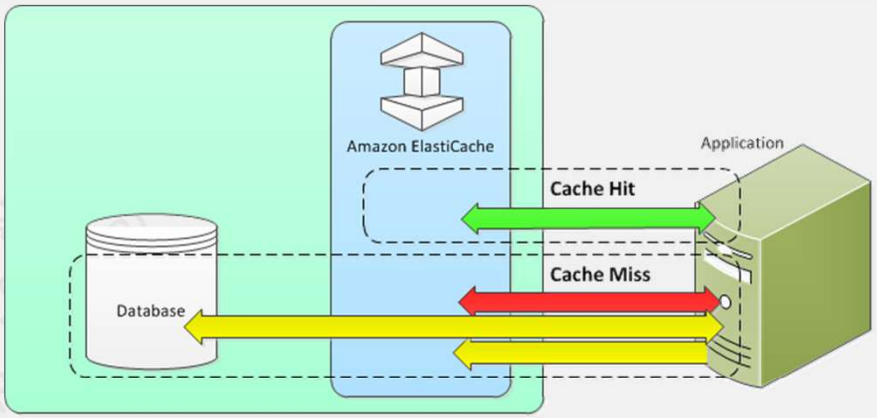
- maintain consistency between caches and the source of truth such as the database through cache invalidation
- Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
- Time to Live (TTL): Expiration time for records in the cache
  - Eventual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



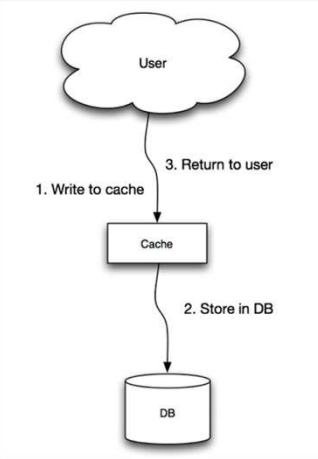

```
def get_user(self, user_id):
    user = cache.get("user.{0}", user_id)
    if user is None:
        user = db.query("SELECT * FROM users WHERE user_id = {0}", user_id)
        if user is not None:
            key = "user.{0}".format(user_id)
            cache.set(key, json.dumps(user))
    return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

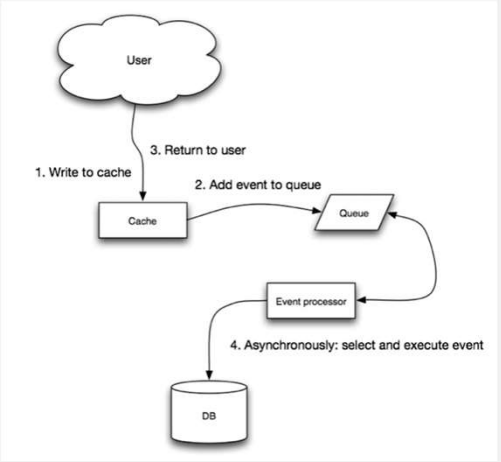
```
def set_user(user_id, values):
    user = db.query("UPDATE Users WHERE id = {0}", user_id, values)
    cache.set(user_id, user)
```

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

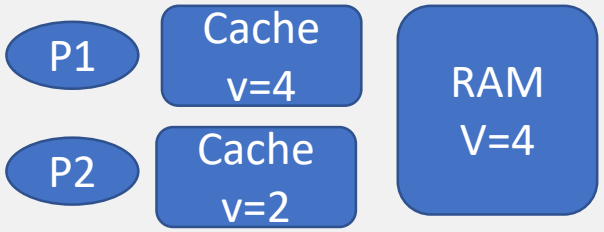
- Since no data is returned to the requester on write operations, a decision needs to be made on write misses, whether or not data would be loaded into the cache. This is defined by these two approaches:
  - *Write allocate* (also called *fetch on write*): data at the missed-write location is loaded 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.

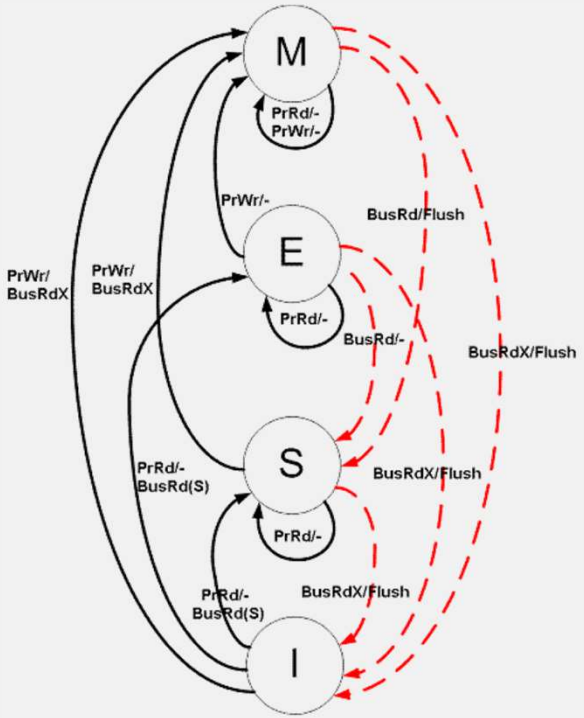
Cache Coherency

- *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



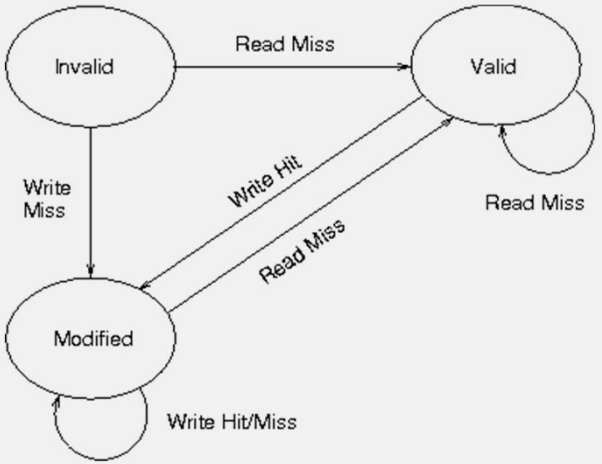
MESI protocol

- 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

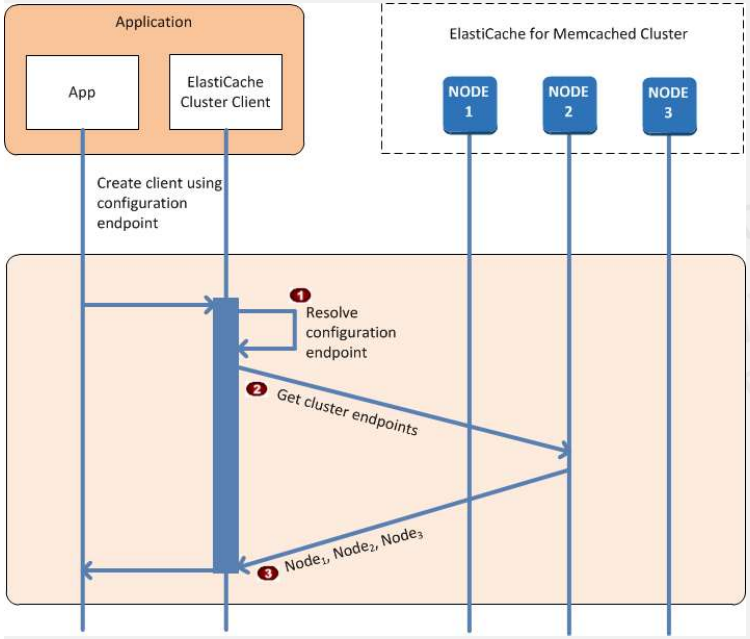
- Simple key/value storage model
  - E.g. in a Lambda Python function, just import pymemcache and set / get data
  - Other libraries: python-Memcached, pylibmc, twisted-Memcached, etc.
- Simple data model:
  - Strings, Objects
  - From the results of database calls, API calls, even HTML page renderings (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

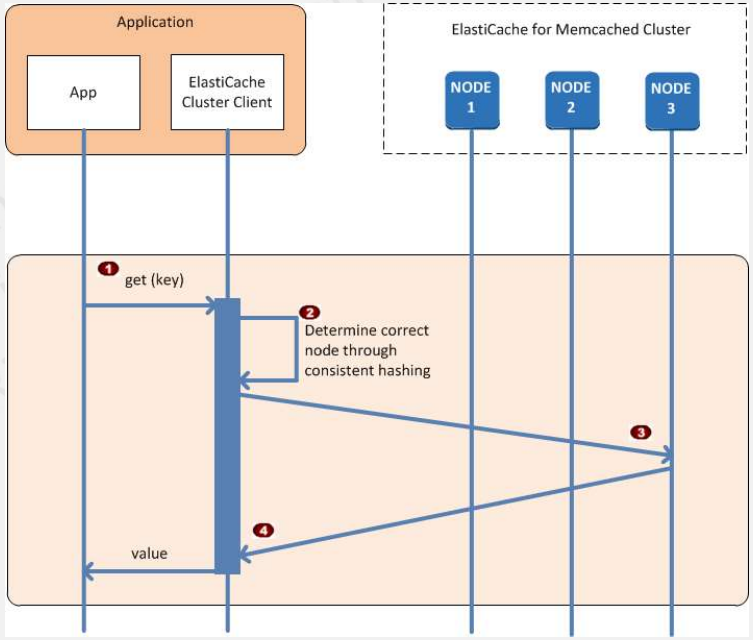
- Configuration endpoint
  - When you increase the number of nodes in a cache cluster, the new nodes register themselves with the configuration endpoint and with all of the other nodes
- When you remove nodes from the cache cluster, the departing nodes deregister themselves.
- In both cases, all the 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")
```

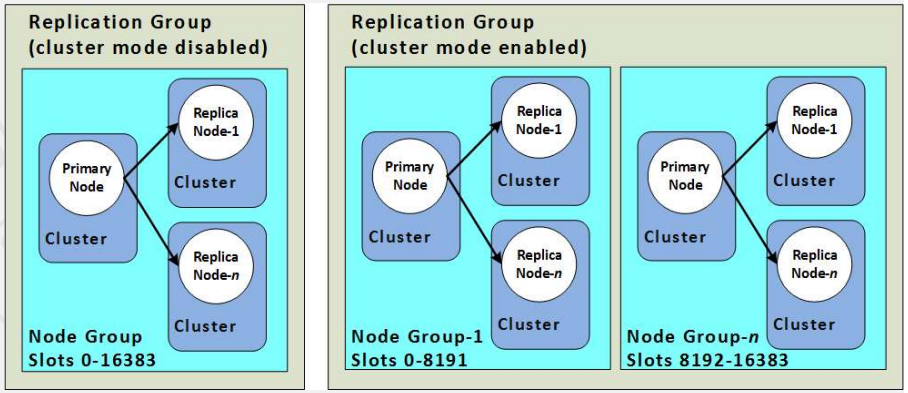
CLOUD COMPUTING APPLICATIONS

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

The Right Caching Strategy

- Cache-Aside (Lazy Loading)
  - Applicating data is written only into the source
  - Only loads data to the cache when it is required on a “read”
    - Typically most data is never requested
  - Suitable for read-heavy Applications
  - Allows stale data
  - In case of cache node failure, just read from source
- Write-Through
  - Applicating data is written into the cache and source at the same time
  - Suitable for write-heavy Applications, where data loss is not acceptable
    - But every write is expensive
  - Cache never gets stale
  - In case of cache node failure, just read from source
- Write-Back (Lazy Writing)
  - Applicating data is written only to the Cache
  - More complex to implement

Cache Sharding

- There is only one machine that contains each piece of data
- Memcached:
  - Consistent Hashing ring algorithm
- Redis:
  - Elasticache for Redis can have up to 250 shards
  - Each shard can consist of a master Redis node, and up to 5 Redis Read replicas
- Sharding is a great technique but has its own problems
  - Resharding data when adding/removing nodes
  - Celebrity problem
  - Join and de-normalization:
    - Not as big a problem in Caches, but can be serious for databases
    - Once data is sharded, it is hard to perform join operations across all shards.

Time to Live (TTL)

- Lazy loading allows for stale data but doesn't fail with empty nodes
- Write-through ensures that data is always fresh, but can fail with empty nodes and can populate the cache with superfluous data
- By adding a time to live (TTL) value to each write, you can have the advantages of each strategy. At the same time, you can and largely avoid cluttering up the cache with extra data.
- *Time to live (TTL)* is an integer value that specifies the number of seconds until the key expires
  - Redis can specify seconds or milliseconds for this value.
  - For Memcached, it is seconds.
- When an application attempts to read an expired key, it is treated as though the key is not found. The database is queried for the key and the cache is updated.
- This approach doesn't guarantee that a value isn't stale. However, it keeps data from getting too stale and requires that values in the cache are occasionally refreshed from the database.

CLOUD COMPUTING APPLICATIONS

Roy Campbell & Reza Farivar

Redis

Redis: REmote DIctionary Server

- Open Source
- Written in C
- Data model is a dictionary which maps keys to values
- Supports not only strings, but also abstract data types:
  - Lists of strings
  - Sets of strings (collections of non-repeating unsorted elements)
  - Sorted sets of strings (collections of non-repeating elements ordered by a floating-point number called score)
  - Hashes where keys and values are strings

Redis

- Ultrafast response time
  - Everything is in memory
  - Non-blocking I/O, single threaded
  - 100,000+ read / writes per second
- Periodically checkpoints in-memory values to disk every few seconds
- In case of failure, only the very last seconds' worth of key/values are lost

Potential Uses

- Session store
  - One (or more) sessions per user
  - Many reads, few writes
  - Throw-away data
  - Timeouts
- Logging
  - Rapid, low latency writes
  - Data you don’t care that much about
  - Not that much data (must be in-memory)

General Cases

- Data that you don’t mind losing
- Records that can be accessed by a single primary key
- Schema that that is either a single value or is a serialized object

Simple Programming Model

- GET, SET, INCR, DECR, EXISTS, DEL
- HGET, HSET, KEYS, HDEL
- SADD, SMEMBERS, SREM
- PUBLISH, SUBSCRIBE

Summary

- Redis is not a database
  - It complements your existing data storage layer
  - E.g. stackoverflow uses Redis for data caching
- Publish/Subscribe support

CLOUD COMPUTING APPLICATIONS

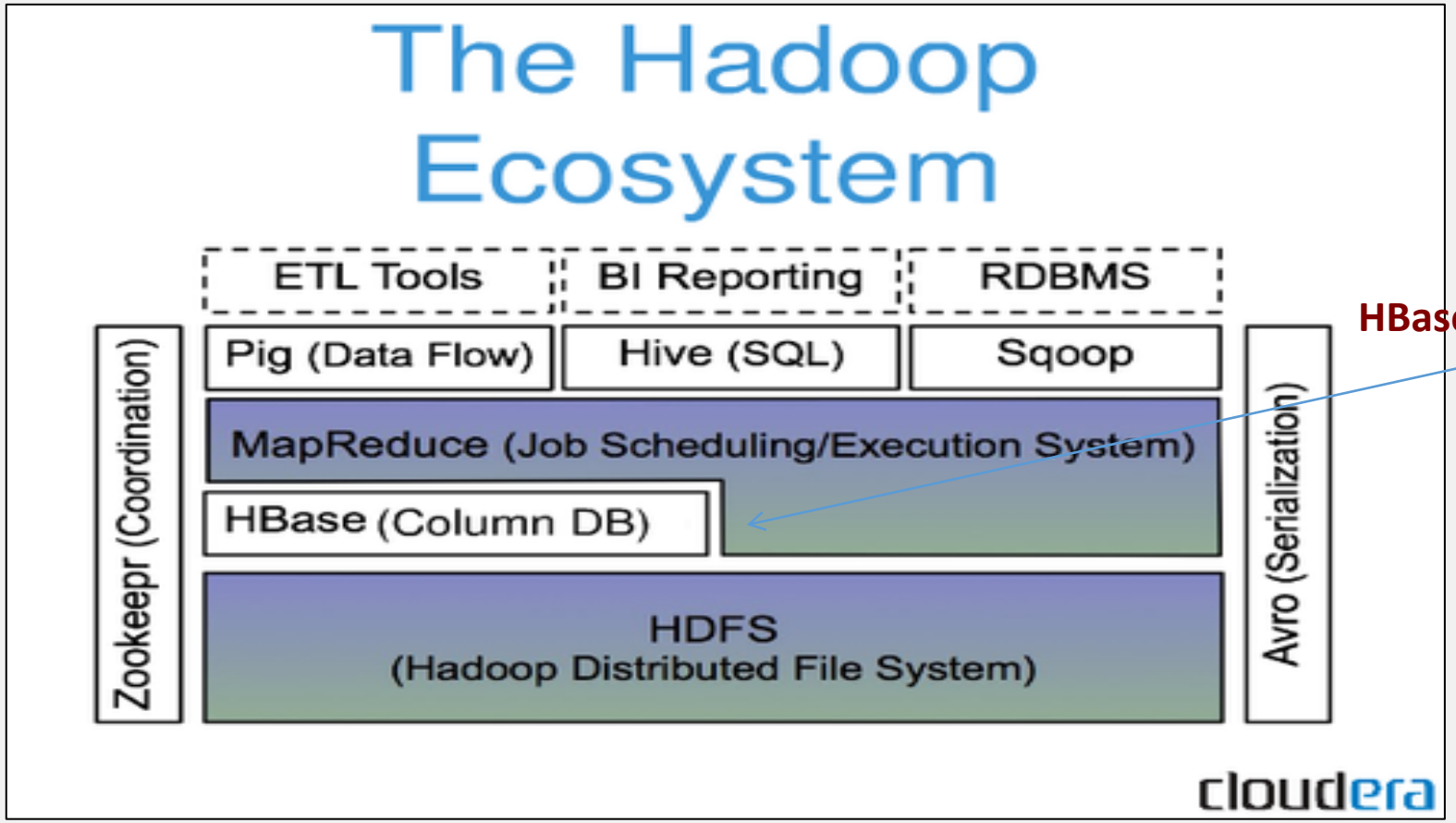
HBase Usage API

Prof. Roy Campbell

HBase: Overview

- HBase is a distributed column-oriented data store built on top of HDFS
- HBase is an Apache open source project whose goal is to provide storage for Hadoop Distributed Computing
- Data is logically organized into tables, rows, and columns

HBase:
Part of Hadoop's Ecosystem



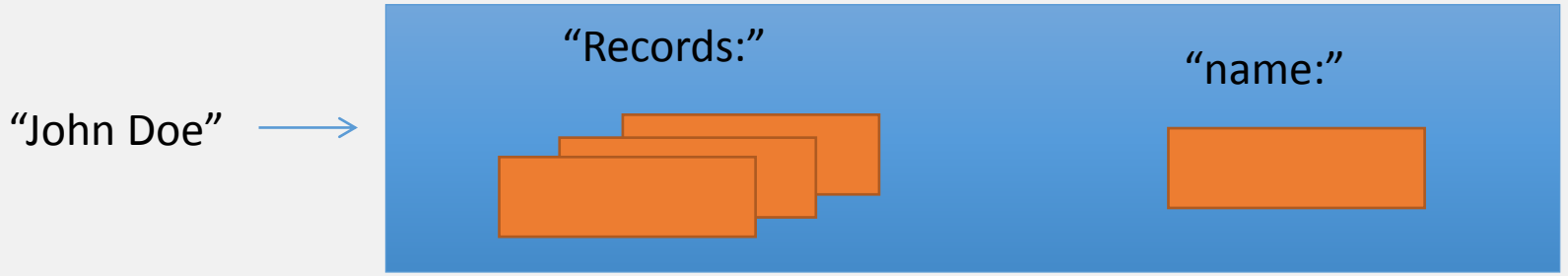
HBase is built on top of HDFS

HBase files are internally stored in HDFS

HBase vs. HDFS

- ***HDFS*** is good for batch processing (scans over big files)
  - Not good for record lookup
  - Not good for incremental addition of small batches
  - Not good for updates
- ***HBase*** addresses the above points
  - Fast record lookup
  - Support for record-level insertion
  - Support for updates

Data Model

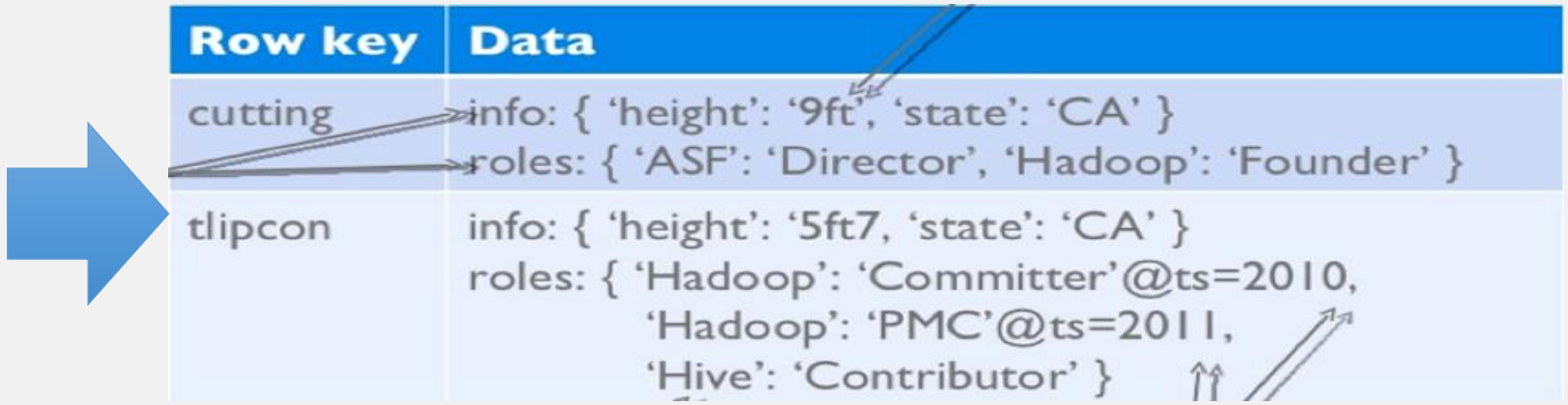


- A table in Bigtable is a sparse, distributed, persistent multidimensional sorted map
- Map indexed by a row key, column key, and a timestamp
  - (row:string, column:string, time:int64) → uninterpreted byte array
- Supports lookups, inserts, deletes
  - Single row transactions only

Notes on Data Model

- HBase schema consists of several ***Tables***
- Each table consists of a set of ***Column Families***
  - Columns are not part of the schema
- HBase has ***Dynamic Columns***
  - Because column names are encoded inside the cells
  - Different cells can have different columns

"Roles" column family has different columns in different cells



Row key	Data
cutting	info: { 'height': '9ft', 'state': 'CA' } roles: { 'ASF': 'Director', 'Hadoop': 'Founder' }
tlipcon	info: { 'height': '5ft7, 'state': 'CA' } roles: { 'Hadoop': 'Committer'@ts=2010, 'Hadoop': 'PMC'@ts=2011, 'Hive': 'Contributor' }

Notes on Data Model

- The ***version number*** can be user-supplied
  - Does not have to be inserted in increasing order
  - Version numbers are unique within each key
- Table can be very sparse
  - Many cells are empty
- ***Keys*** are indexed as the primary key

Has two columns
[cnnsi.com & my.look.ca]

Row Key	Time Stamp	ColumnFamily `contents`	ColumnFamily `anchor`
"com.cnn.www"	t9		`anchor:cnnsi.com` = "CNN"
"com.cnn.www"	t8		`anchor:my.look.ca` = "CNN.com"
"com.cnn.www"	t6	`contents:html` = "<html>..."	
"com.cnn.www"	t5	`contents:html` = "<html>..."	
"com.cnn.www"	t3	`contents:html` = "<html>..."	

Rows and Columns

- Rows maintained in sorted lexicographic order
  - Applications can exploit this property for efficient row scans
  - Row ranges dynamically partitioned into tablets
- Columns grouped into column families
  - Column key = *family:qualifier*
  - Column families provide locality hints
  - Unbounded number of columns

HBase lookup example

```
Scanner scanner (T);
ScanStream *stream;
stream = scanner.FetchColumnFamily(“name”);
stream -> SetReturnAllVersions();
//Filter return sets using regex
scaanner.lookup (“John Doe”);
for (; !stream -> Done(); stream -> Next()) {
   printf (“%s %s %s \n”,
          scanner.RowName(),
          stream -> ColumnName(),
          stream – Value());
}
```

HBase lookup example

```
Table *T = OpenOrDie (“/hbase/myTable”);

RowMutation rowMut (T, “John Doe”);
rowMut.Set (“name:Jane Roe”, “NAMES”);
rowMut.Delete(“name:Jack Public”);
Operation op;
Apply (&op, &rowMut);
```

CLOUD COMPUTING APPLICATIONS

HBase Internals - Part 1

Roy Campbell & Reza Farivar

HBase

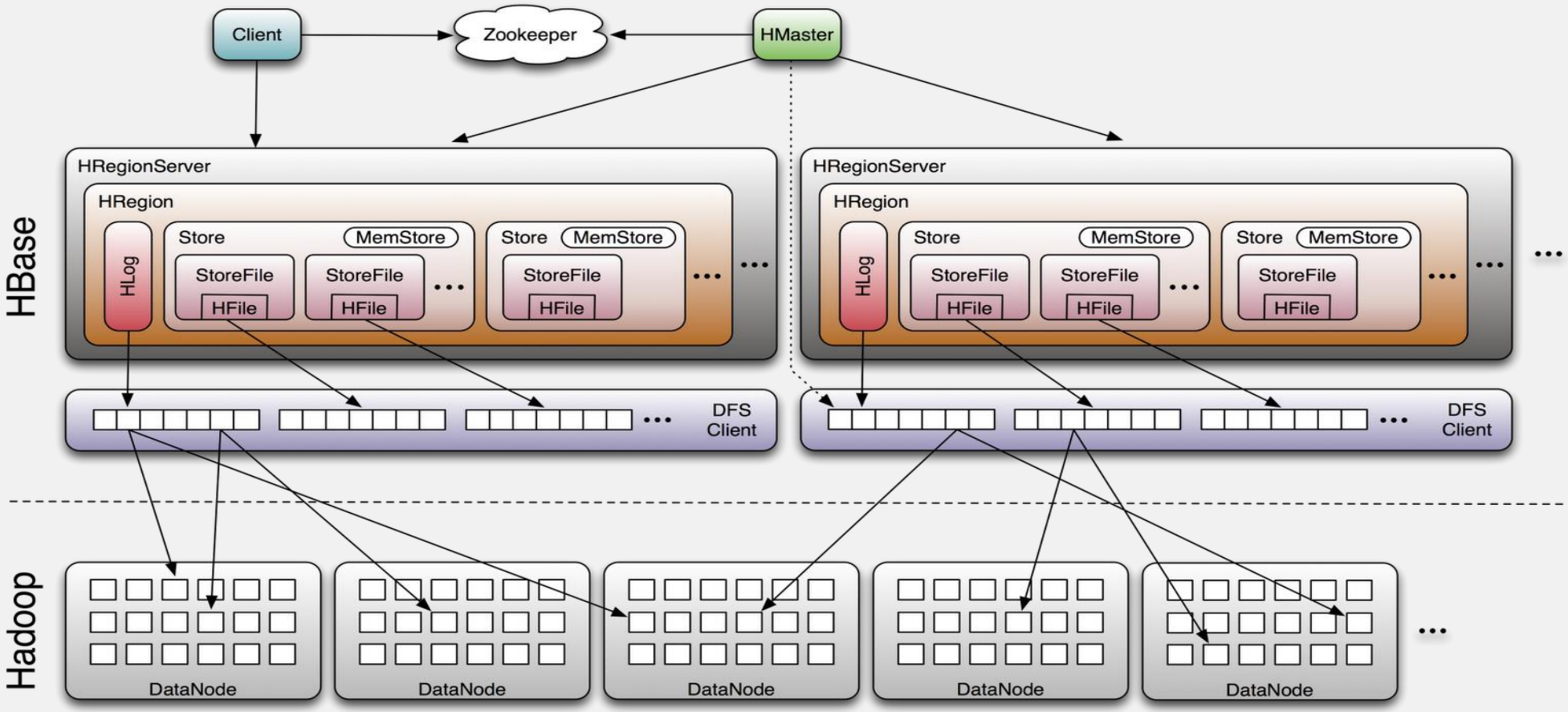


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

HBase Building Blocks

- HDFS
- Apache ZooKeeper
  - ZooKeeper uses ZAB (ZooKeeper's Atomic Broadcast)
- HFile

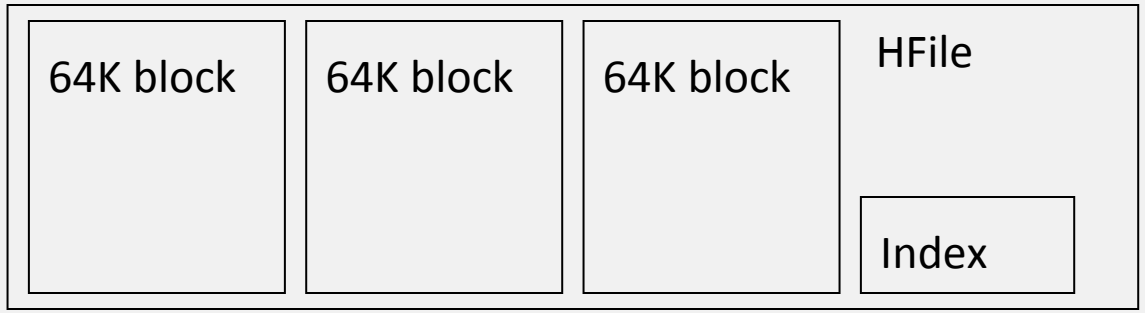
HFile

- Basic building block of HBase
- On-disk file format representing a map from string to string
- Persistent, ordered immutable map from keys to values
  - Stored in HDFS

Source: Graphic from slides by Erik Paulson

HFile

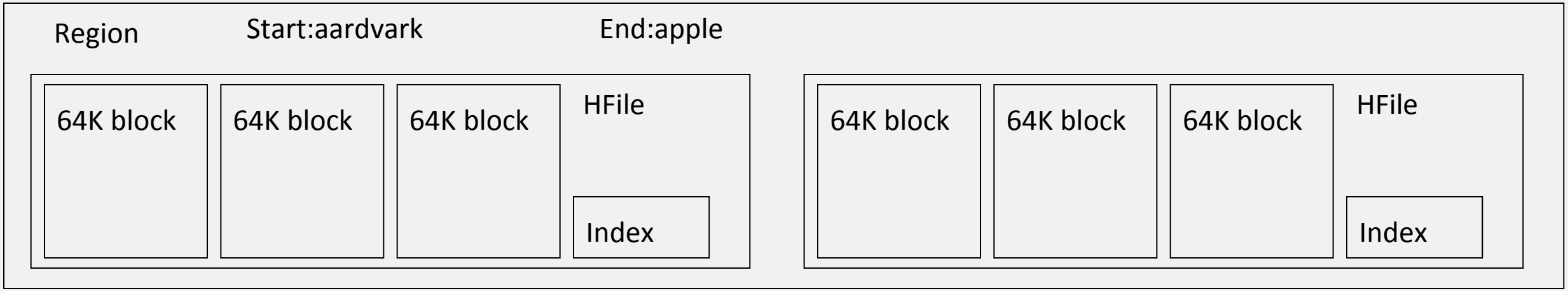
- Sequence of blocks on disk plus an index for block lookup
  - Can be completely mapped into memory
  - MemStore
- Supported operations:
  - Lookup value associated with key
  - Iterate key/value pairs within a key range



Source: Graphic from slides by Erik Paulson

HRegion

- Dynamically partitioned range of rows
- Built from multiple HFiles



Source: Graphic from slides by Erik Paulson

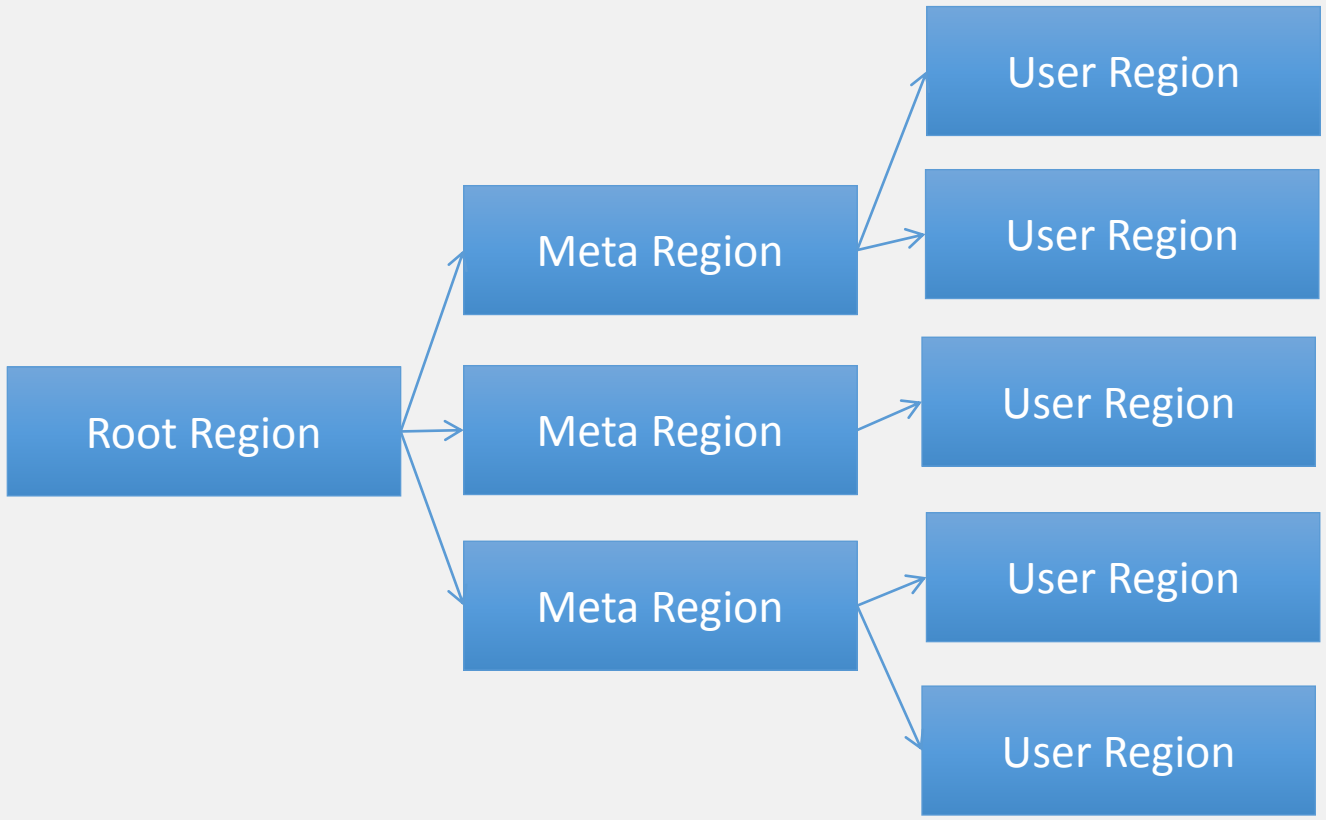
HBase Master

- Assigns HFiles to HRegion servers
- Detects addition and expiration of HRegion servers
- Balances HRegion server load. HRegions are distributed randomly on nodes of the cluster for load balancing.
- Handles garbage collection
- Handles schema changes

HRegion Servers

- Each HRegion server manages a set of regions
  - Typically between 10 to 1,000 regions
  - Each 100-200 MB by default
- Handles read and write requests to the regions
- Splits regions that have grown too large

HRegion Location



CLOUD COMPUTING APPLICATIONS

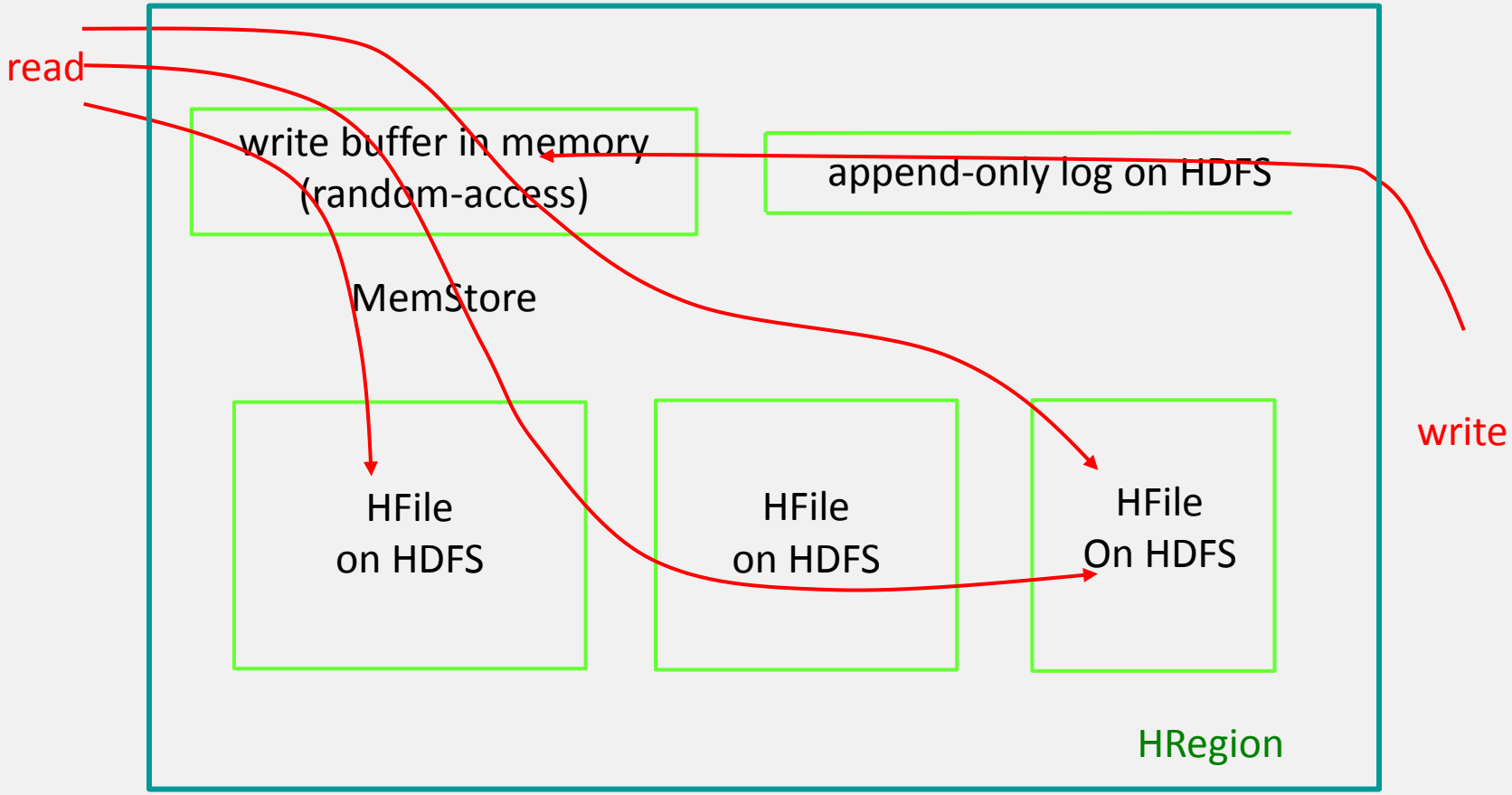
HBase Internals - Part 2

Roy Campbell & Reza Farivar

HRegion Assignment

- HBase Master keeps track of:
  - Set of live HRegion servers
  - Assignment of HRegions to HRegion servers
  - Unassigned HRegions
- Each HRegion is assigned to one HRegion server at a time
  - HRegion server maintains an exclusive lock on a file in ZooKeeper
  - HBase Master monitors HRegion servers and handles assignment
- Changes to HRegion structure
  - Table creation/deletion (HBase Master initiated)
  - HRegion merging (HBase Master initiated)
  - HRegion splitting (HRegion server initiated)

HRegion in Memory Representation



CLOUD COMPUTING APPLICATIONS

Roy Campbell & Reza Farivar

Spark SQL

Spark SQL

- Structured Data Processing in Apache Spark
- Built on top of RDD data abstraction
- Need more information about the columns (Schema)
  - Can use this for optimizations
- Spark can read data from HDFS, Hive tables, JSON, etc.
- Can use SQL to query the data
- When needed, switch between SQL and python/java/scala
- Strong query engine

DataSet

- A Dataset is a distributed collection of data
- Dataset provides the benefits of RDDs (strong typing, ability to use powerful lambda functions) with the benefits of Spark SQL's optimized execution engine
- A Dataset can be constructed from JVM objects and then manipulated using functional transformations (map, flatMap, filter, etc.)
- The Dataset API is available in Scala and Java
  - Python does not currently (as of 2.0.0) have the support for the Dataset API.
  - But due to Python's dynamic nature, many of the benefits of the Dataset API are already available (row.columnName).

DataFrame

- A DataFrame is a *Dataset* organized into named columns
- Equivalent to a table in a relational database or a data frame in R/Python
- DataFrames can be constructed from:
  - structured data files
  - tables in Hive
  - external databases
  - existing RDDs
- The DataFrame API is available in Scala, Java, Python, and R

CLOUD COMPUTING APPLICATIONS

HIVE

Roy Campbell & Reza Farivar

Hive: Background

- Started at Facebook
- Data was collected by nightly cron jobs into Oracle DB
- “ETL” via hand-coded Python
- HQL, a variant of SQL
- Translates queries into map/reduce jobs, Hadoop YARN, Tez, or Spark
- Note
  - No UPDATE or DELETE support, for example
  - Focuses primarily on the query part of SQL

Hive: Example

- Hive looks similar to an SQL database
- Relational join on two tables:
  - Table of word counts from Shakespeare collection
  - Table of word counts from Homer

Hive: Example

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN homer k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;
```

the	25848	62394
I	23031	8854
and	19671	38985
to	18038	13526
of	16700	34654
a	14170	8057
you	12702	2720
my	11297	4135
in	10797	12445
is	8882	6884

Hive: Behind the Scenes

```
SELECT s.word, s.freq, k.freq FROM shakespeare s
  JOIN homer k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
  ORDER BY s.freq DESC LIMIT 10;
```

(Abstract Syntax Tree)

```
(TOK_QUERY (TOK_FROM (TOK_JOIN (TOK_TABREF shakespeare s) (TOK_TABREF homer k) (= (.
(TOK_TABLE_OR_COL s) word) (. (TOK_TABLE_OR_COL k) word)))) (TOK_INSERT (TOK_DESTINATION (TOK_DIR
TOK_TMP_FILE)) (TOK_SELECT (TOK_SELEXPR (. (TOK_TABLE_OR_COL s) word)) (TOK_SELEXPR (.
(TOK_TABLE_OR_COL s) freq)) (TOK_SELEXPR (. (TOK_TABLE_OR_COL k) freq))) (TOK_WHERE (AND (>= (.
(TOK_TABLE_OR_COL s) freq) 1) (>= (. (TOK_TABLE_OR_COL k) freq) 1))) (TOK_ORDERBY
(TOK_TABSORTCOLNAMEDESC (. (TOK_TABLE_OR_COL s) freq))) (TOK_LIMIT 10)))
```

(one or more of MapReduce jobs)

Hive Components

- Shell: allows interactive queries
- Driver: session handles, fetch, execute
- Compiler: parse, plan, optimize
- Execution engine: DAG of stages (MR, HDFS, metadata)
- Metastore: schema, location in HDFS, etc.

Metastore

- Hive uses a traditional database to store its metadata
  - A namespace containing a set of tables
  - Holds table definitions (column types, physical layout)
  - Holds partitioning information
- Can be stored in MySQL, Oracle, and many other relational databases

Physical Layout

- Warehouse directory in HDFS
  - E.g., /user/hive/warehouse
- Tables stored in subdirectories of warehouse
  - Partitions form subdirectories of tables
- Actual data stored in flat files
  - Control char-delimited text, or SequenceFiles
  - Can be customized to use arbitrary format

CLOUD COMPUTING APPLICATIONS

Decoupling in Cloud Architectures

Prof. Reza Farivar

Multi-tier Distributed Architecture

- Enterprise architectures require elasticity and scalability
  - Scalability: respond to increasing demand
  - Elasticity: respond to decreasing demand
- Fault tolerance
  - Component failure is the rule in cloud environments
- Changing demand patterns
  - Hard to predict how many resources we will need in the future
- Complexity
  - Multiple platforms and development teams

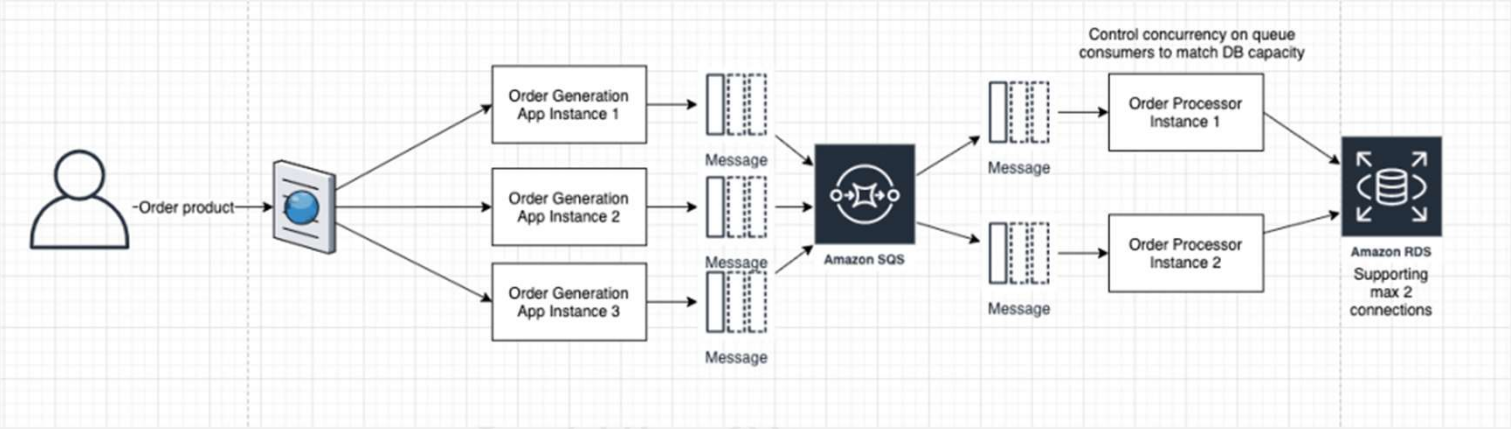
Decoupling

- The key to achieving reliable, scalable, elastic architectures is decoupling
- Building applications from individual components that each perform a discrete function
- A reliable queue between components
  - Allows many integration patterns for connecting services
- Loose coupling → increases an architecture's resiliency to failure and ability to handle traffic spikes
  - producer and consumer operate independently
- Asynchronous Communication

Message Queues

- Message queues → decouple and scale microservices, distributed systems, and serverless applications
  - Send, store, and receive messages between software components
  - Any volume
  - Without losing messages
  - No need to rely on other services be always available
- Can easily handle momentary spikes in demands
  - Up spikes and down spikes
- Guaranteed message delivery
  - At least Once
  - Exactly Once

Example



https://aws.amazon.com/blogs/architecture/application-integration-using-queues-and-messages/

Message Queue Platforms

- Open Source
  - Apache ActiveMQ
  - Apache RabbitMQ
  - Apache Kafka
- Proprietary / cloud services
  - Amazon AWS Simple Queue Service
  - Amazon MQ
    - Apache ActiveMQ
    - Apache RabbitMQ
  - Amazon Kinesis
  - Amazon Managed Kafka

Publish-Subscribe Model

- A sibling of the message queue systems
- Producers publish messages to the queue
- Several consumers, having subscribed to a specific producer (or topic, etc.), all receive the message
- Publishers and subscribers are decoupled
- Example:
  - Kafka
  - AWS Simple Notification Service

CLOUD COMPUTING APPLICATIONS

Amazon Simple Queue Service

Prof. Reza Farivar

AWS SQS

- Simple Queue Service
- **put-get-delete** paradigm
  - It requires a consumer to explicitly state that its data has finished processing the message it pulled from the queue
  - The message data is kept safe with the queue and gets deleted from the queue only after confirmation that it has been processed
- Multiple Producers and Consumers
- Pull model
  - The consumers must explicitly pull the queue
- Redundant infrastructure

Guaranteed message delivery: At Least Once

- when a process retrieves the message from the queue, it temporarily makes this message invisible
- When the client informs the queue that it has finished processing the message, SQS deletes this message from the queue
- If the client does not respond back to the queue in a specific amount of time, SQS makes it visible again
- Generally in order message delivery

Guaranteed message delivery: Exactly Once*

- * only true under limited conditions
- FIFO
  - queues work in conjunction with the publisher APIs to avoid introducing duplicate messages
  - Content-based deduplication:
    - SQS generates an SHA-256 hash of the body of the message to generate the content-based message deduplication ID
    - When an application sends a message to the FIFO queue with a message deduplication ID, it is used to deduplicate
  - Message order guarantee
- Bandwidth limits

Recommended reading: https://sookocheff.com/post/messaging/dissecting-sqs-fifo-queues/

Short Pull vs Long pull

- Short Pull: Returns empty if there is nothing in the queue
- What then? Pull again
  - AWS charges based on number of requests
- Solution: Long pull
  - `ReceiveMessageWaitTime`
    - The maximum amount of time that a long-polling receive call waits for a message to become available before returning an empty response

Handling Failures

- Visibility of messages in the Queue
  - A consumer has to explicitly “delete” a message after processing is done
  - Timeout period → message becomes visible again
- What if there is something wrong with a particular message?
  - Infinite loop of visible-invisible?
  - Dead Letter Queue (DLQ)
    - Store failed messages that are not successfully consumed by consumer processes
    - Isolate unconsumed or failed messages and subsequently troubleshoot these messages to determine the reason for their failures
  - Redrive Policy
    - If a queue fails to process a message a predefined number of times, that message is moved to the DLQ

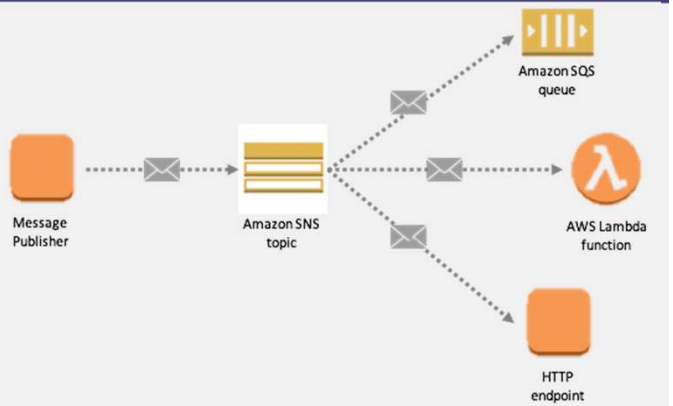
CLOUD COMPUTING APPLICATIONS

Amazon Simple Notification Service

Prof. Reza Farivar

Amazon SNS

- SNS: Simple Notification Service
- PubSub model
  - push
- One to many
- Fan-out architecture
- Endpoint Protocols
  - SQS
  - Lambda
  - HTTP, HTTPs endpoint
  - Email
  - SMS
  - Mobile Device Push Notification



SNS Topic

- A publisher sends a message to an SNS topic
- Subscribers subscribe to SNS topics
- Each SNS topic can have multiple subscribers
  - Each subscriber may use the same protocols or different protocols
- SNS will “push” messages to all sbscribers

SNS Messages

- Subject
- Time to live (TTL)
  - It specifies the time to expire for a message
  - Within a specified time, **Apple Push Notification Service** (**APNS**) or **Google Cloud Messaging** (**GCM**) must deliver messages to the endpoint.
  - If the message is not delivered within the specified time frame, the message will be dropped with no further attempts to deliver the message
- Payload
  - Can be the same, or different for each endpoint protocol type

Managing Access

- AWS uses a mechanism based on policies
  - Policy: JSON document, consisting of statements
  - Statement: describes a single permission written in an access policy language
    - In JSON
- E.g. the user with AWS account 1111-2222-3333 can publish messages to the topic action (for example, Publish)

```
{
  "Statement": [{
    "Sid": "grant-1234-publish",
    "Effect": "Allow",
    "Principal": {
      "AWS": "111122223333"
    },
    "Action": ["sns:Publish"],
    "Resource": "arn:aws:sns:us-east-2:444455556666:MyTopic"
  }]
}
```

CLOUD COMPUTING APPLICATIONS

Kafka

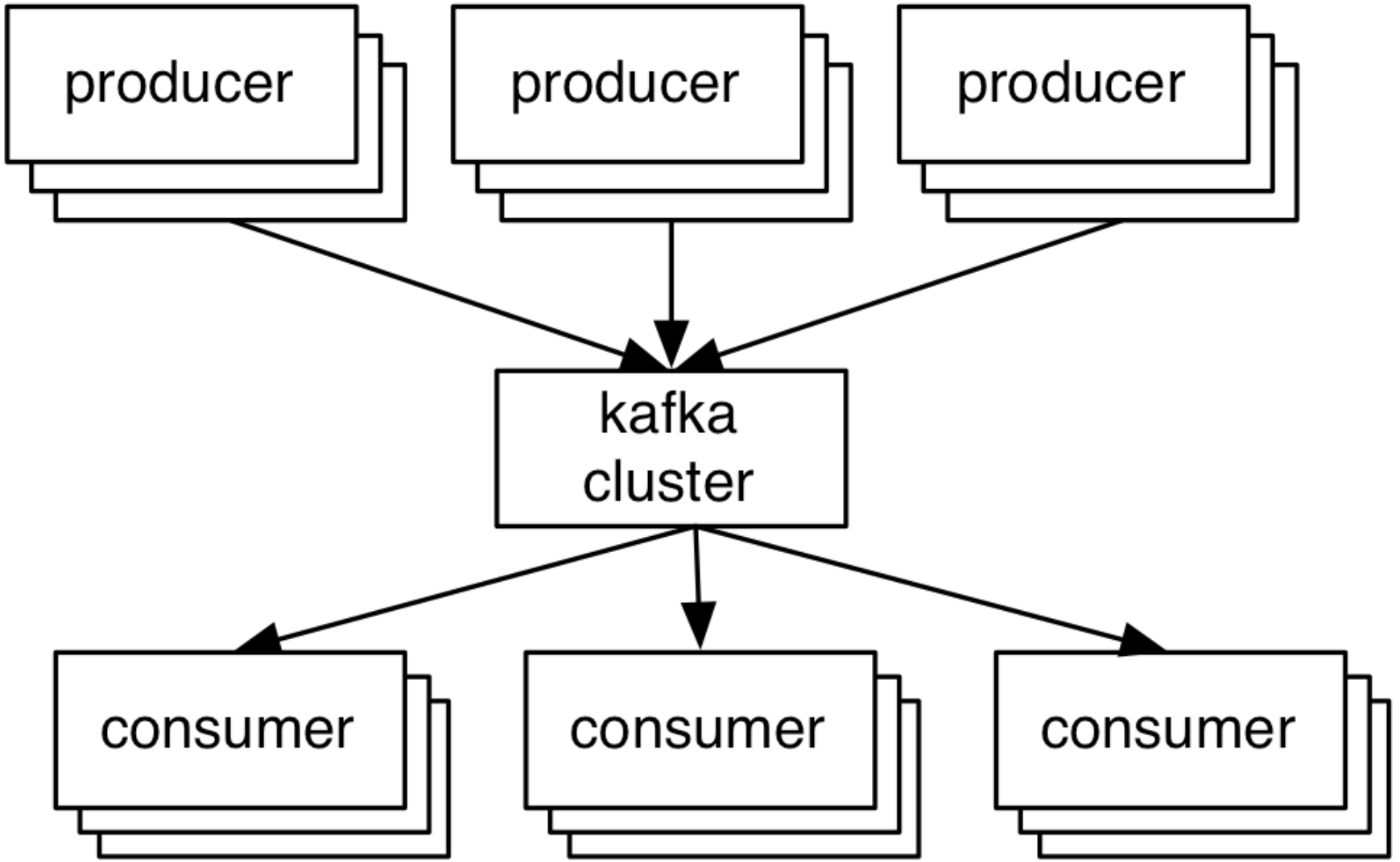
Roy Campbell & Reza Farivar

Thanks to public domain slides Jiangjie (Becket) Qin

Contents

- What is Kafka
- Key concepts
- Kafka clients

Kafka: a distributed, partitioned, replicated publish subscribe system providing commit log service



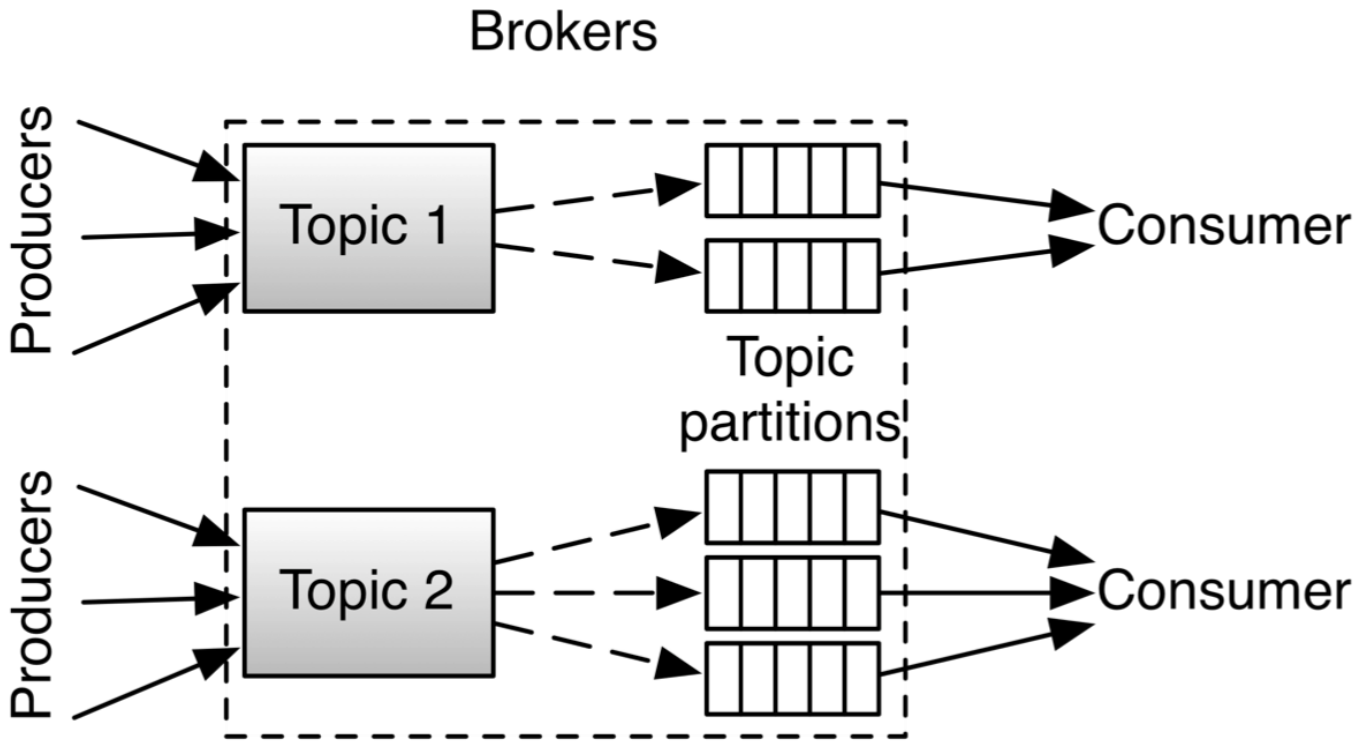
Description

- Kafka maintains feeds of messages in categories called *topics*.
- Processes that publish messages to a Kafka topic are *producers*.
- Processes that subscribe to topics and process the feed of published messages are *consumers*.
- Kafka is run as a cluster comprised of one or more servers each of which is called a *broker*.
- Communication uses TCP, Clients include Java

Characteristics

- Scalability (Kafka is backed by file system)
  - Hundreds of MB/sec/server throughput
  - Many TB per server
- Strong guarantees about messages
  - Strictly ordered (within partitions)
  - All data persistent
- Distributed
  - Replication
  - Partitioning model

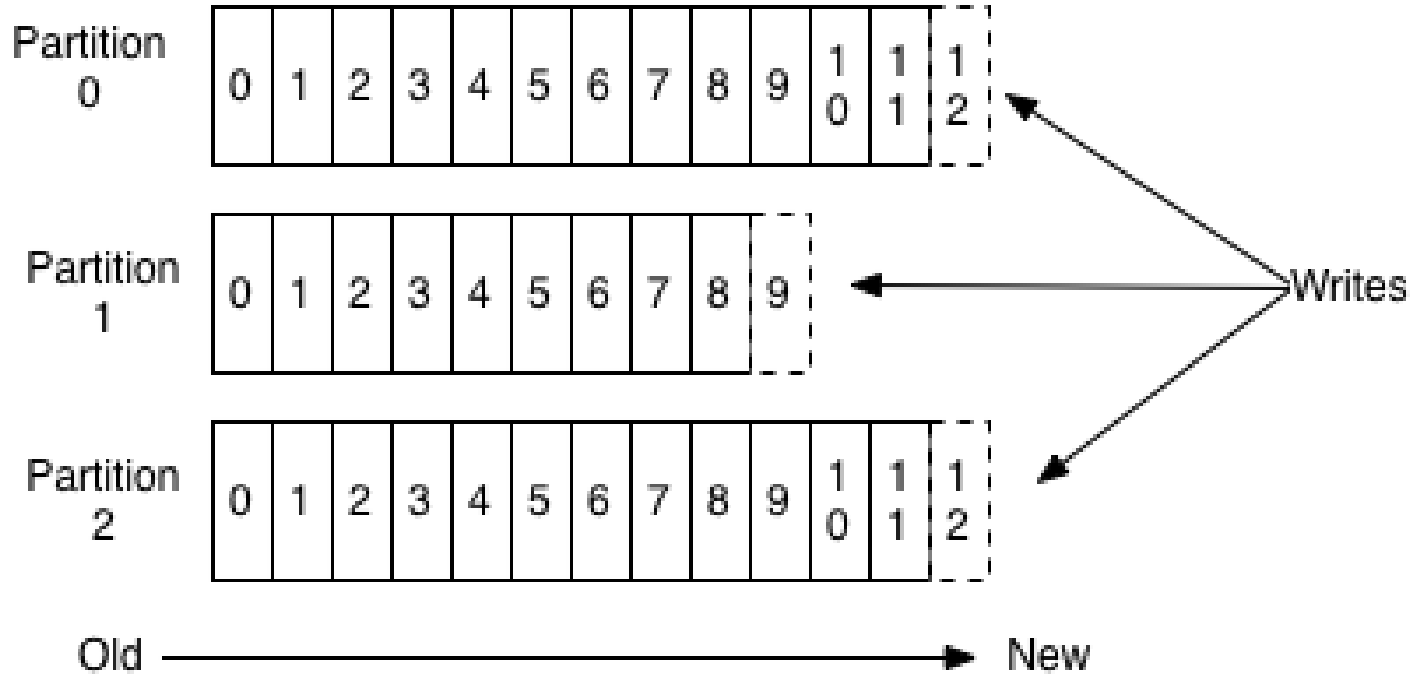
Topics



- A **Topic** has several **Partitions**
- **Partitions** of a **Topic** are distributed across **Brokers**

Topics and Logs

- Kafka store messages about a topic in a partition as an append only log.

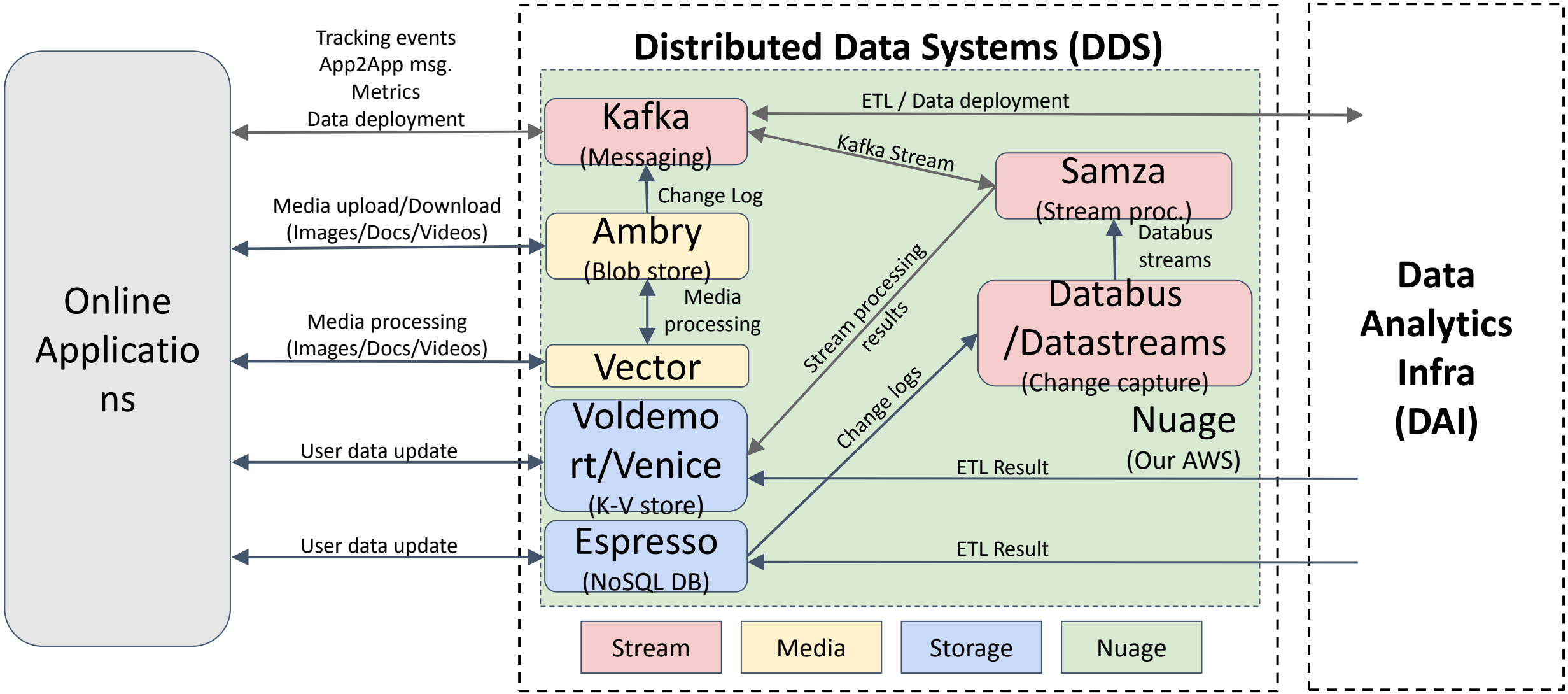


Each partition is an ordered, numbered, immutable append only sequence of messages--- like a commit log.

Kafka Server Cluster Implementation

- Each partition is replicated across a configurable number of servers.
- Each partition has one “leader” server and 0 or more followers.
- A leader handles read and write requests
- A follower replicates the leader and acts as backup.
- Each server is a leader for some of its partitions and a follower for others to load balance
- Zookeeper is used to keep the servers consistent

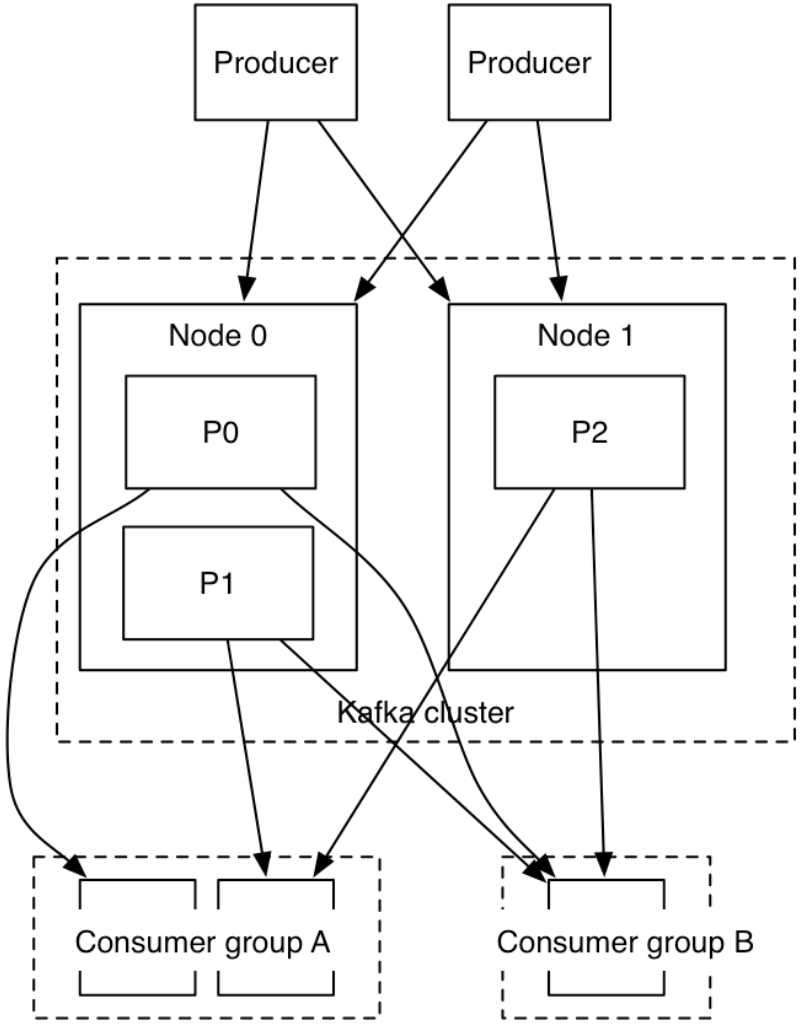
Kafka in a big picture (Linked In)



Producer in Kafka

- Send messages to Kafka Brokers
- Messages are sent to a Topic
  - Messages with same **Key** go to same partition (so they are in order)
  - Messages without a key go to a random partition (no order guarantee here)
  - Number of partitions changed? - Sorry…Same key might go to another partition...

Consumer in Kafka



- A consumer **can** belong to a **Consumer Group (CG)**
- Consumers in the same **CG**
  - Coordinate with each other to determine which consumer will consume from which partition
  - Share the **Consumer Offsets**

Offset

From Brokers' View

- The Index of a message in a log
- **Message Offset** does not change

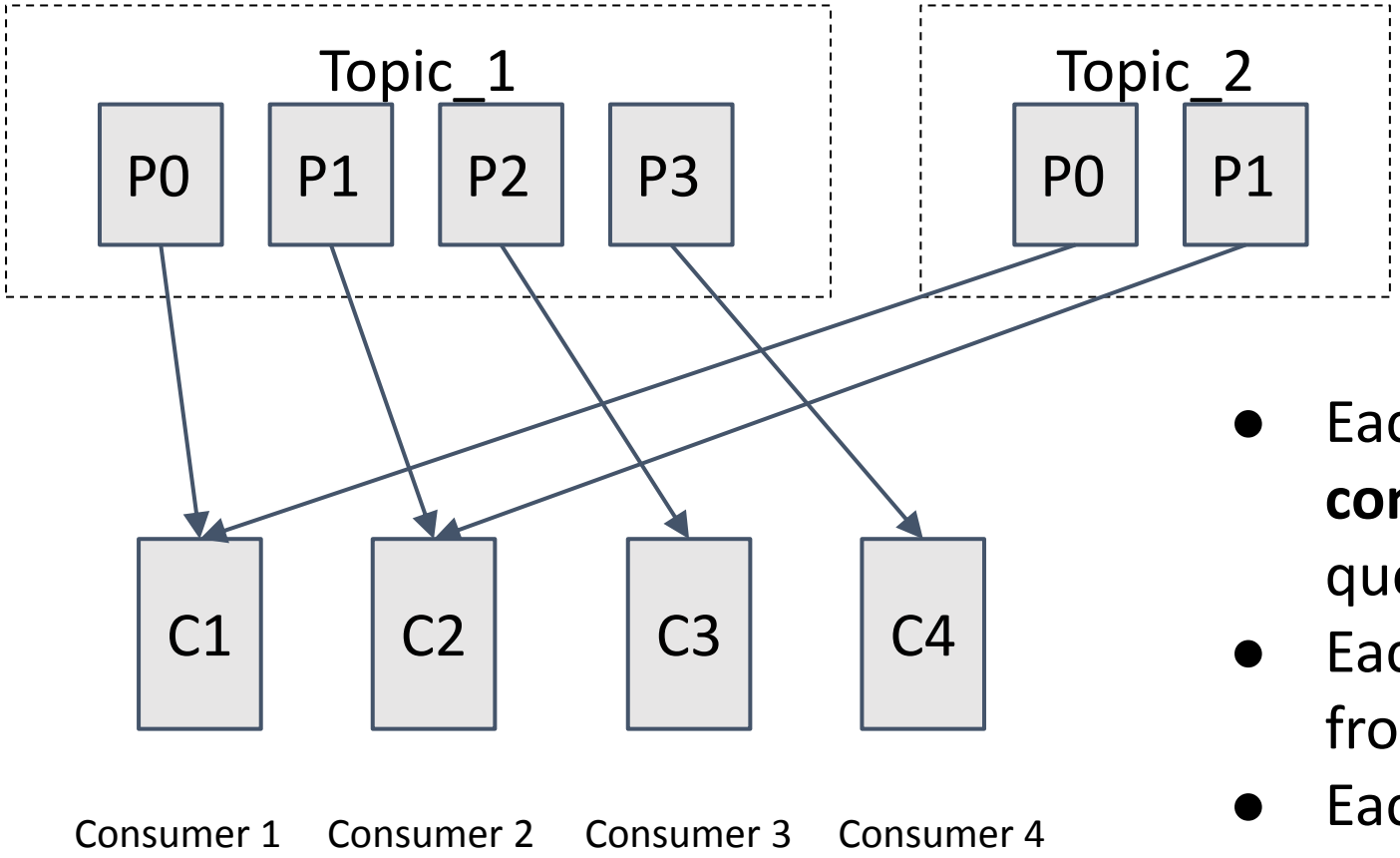
From Consumers' View

- **Consumer Offset**
- The position from where I am consuming
- Consumer Offset can change

More about Consumer Offsets

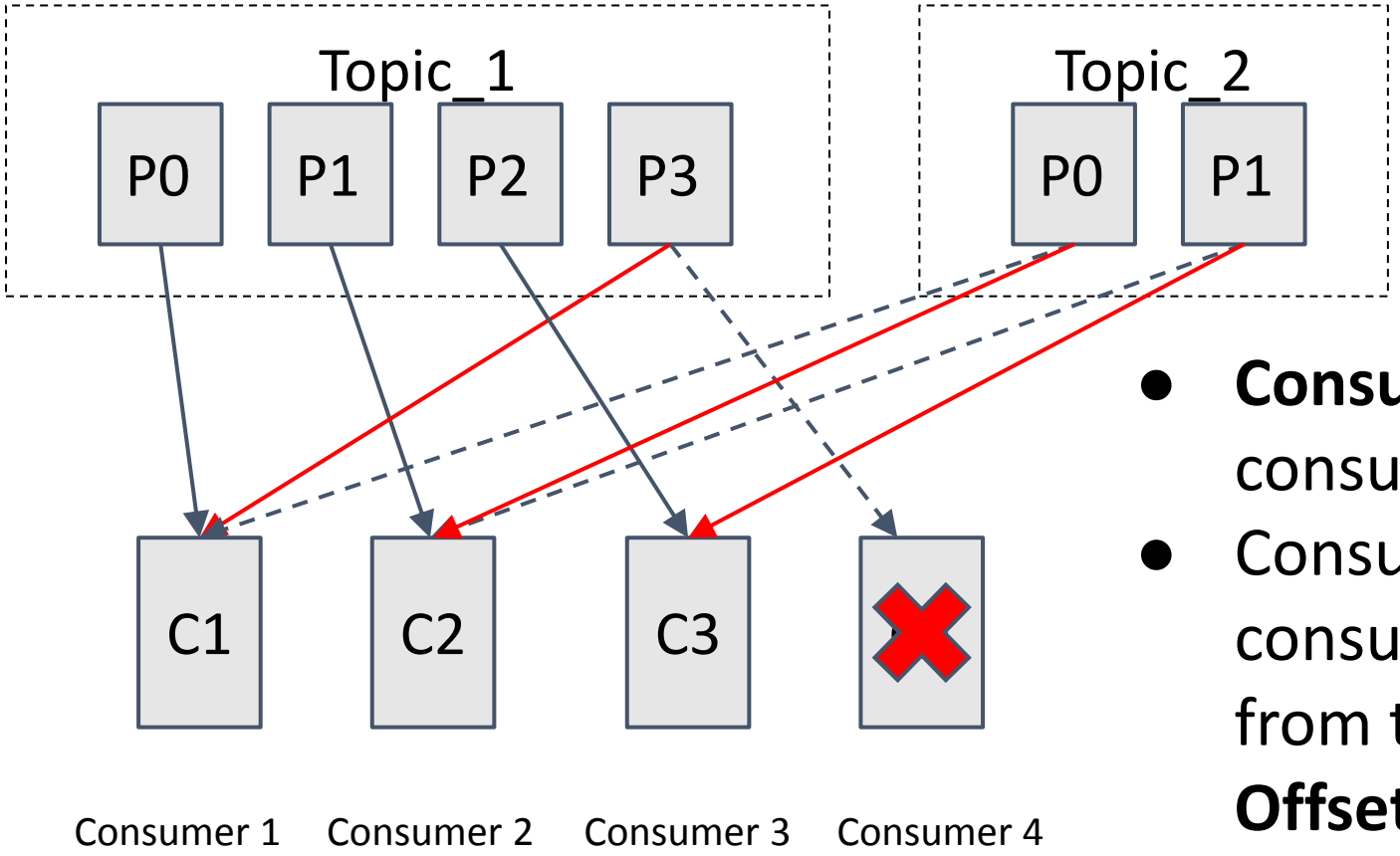
- Consumer Offsets are per
  Topic/Partition/ConsumerGroup
  (For a given group, look up the last consumed position in a topic/partition)
- Consumer Offsets can be **committed** as a checkpoint of consumption so it can be used when
  - Another consumer in the same CG takes over the partition
  - Resuming consumption later from committed offsets

Consumer Rebalance



- Each consumer can have several **consumer threads** (essentially one queue per thread)
- Each consumer thread can consume from multiple partitions
- Each partition will be consumed by **exactly one** consumer **in the entire group**

Consumer Rebalance



- **Consumer rebalance** occurs when consumer 4 is down
- Consumer 1, 2, 3 takes over consumer 4's partitions and **resume** from the last committed **Consumer Offsets of the CG**
- **Transparent to user**