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

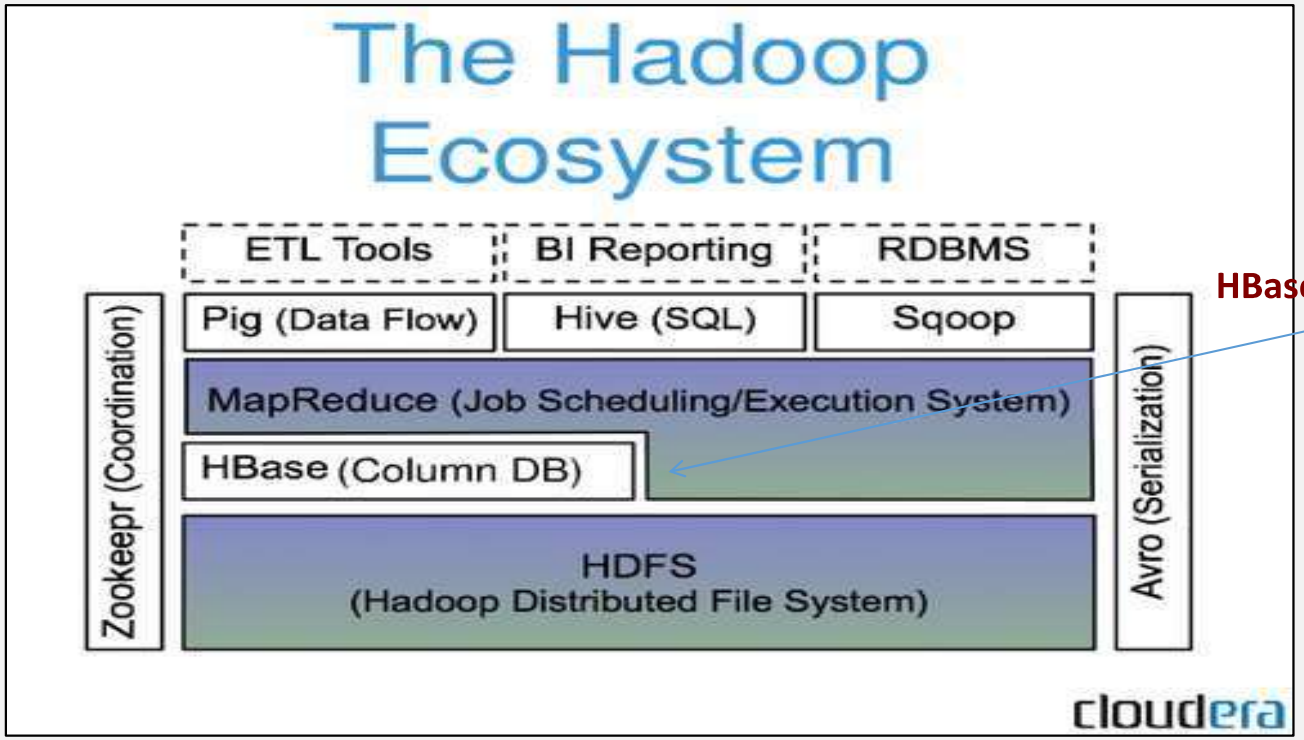
HBase Usage API

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



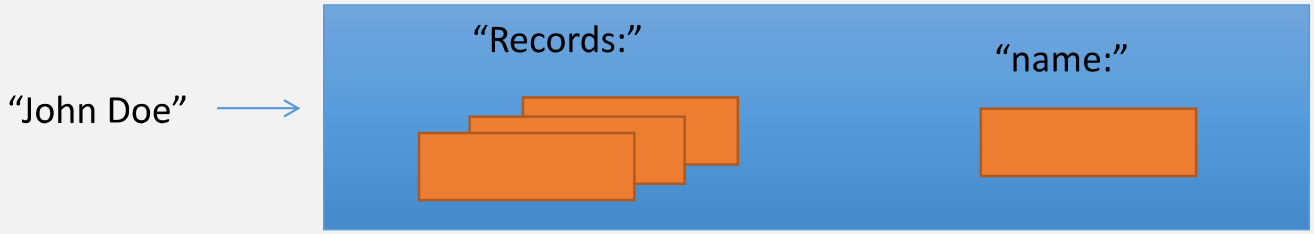
HBase is built on top of HDFS

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

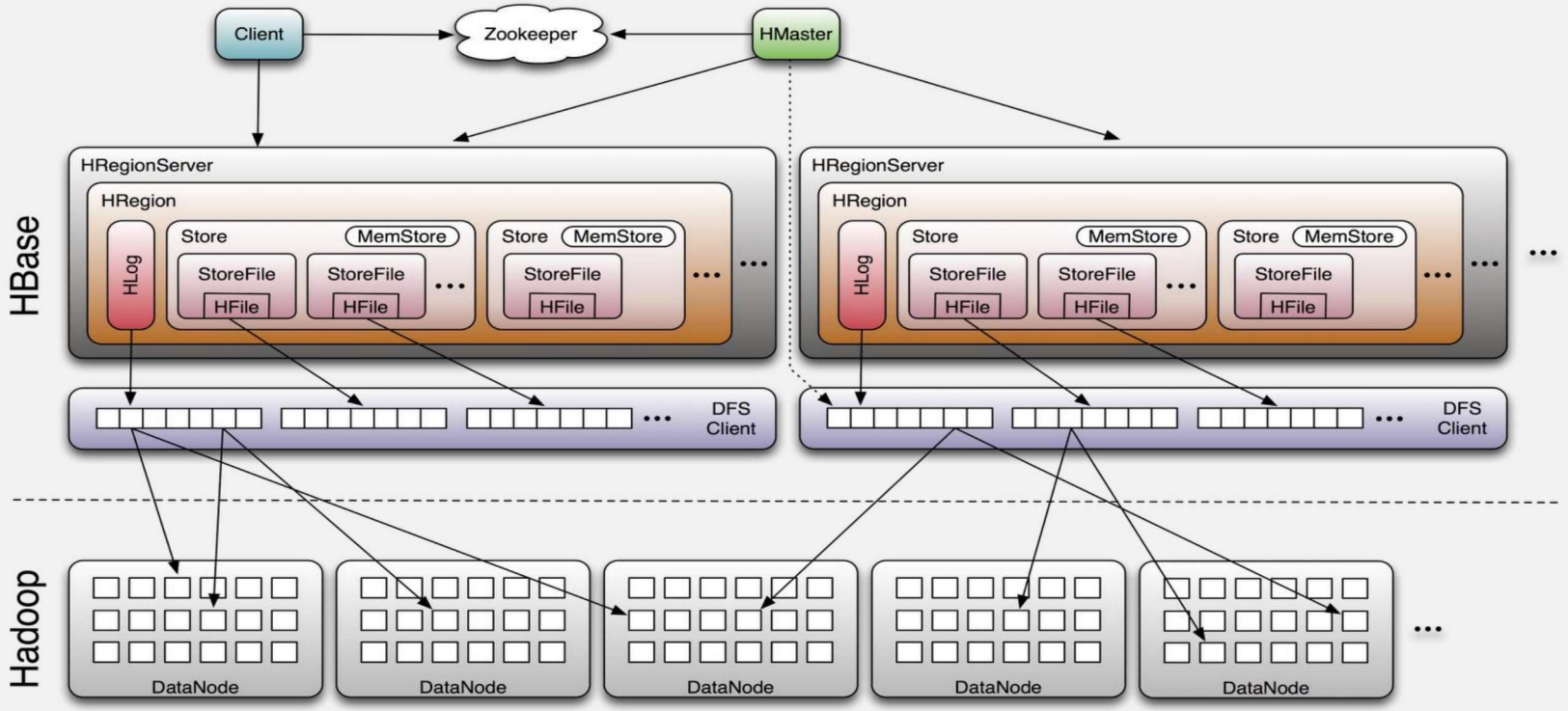


Image Source: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

HBase Building Blocks

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

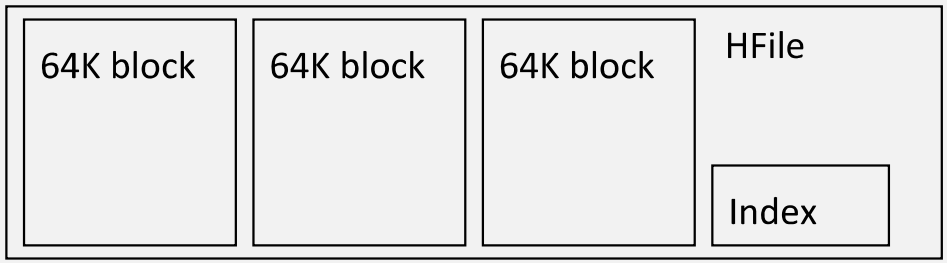
HFile

- 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

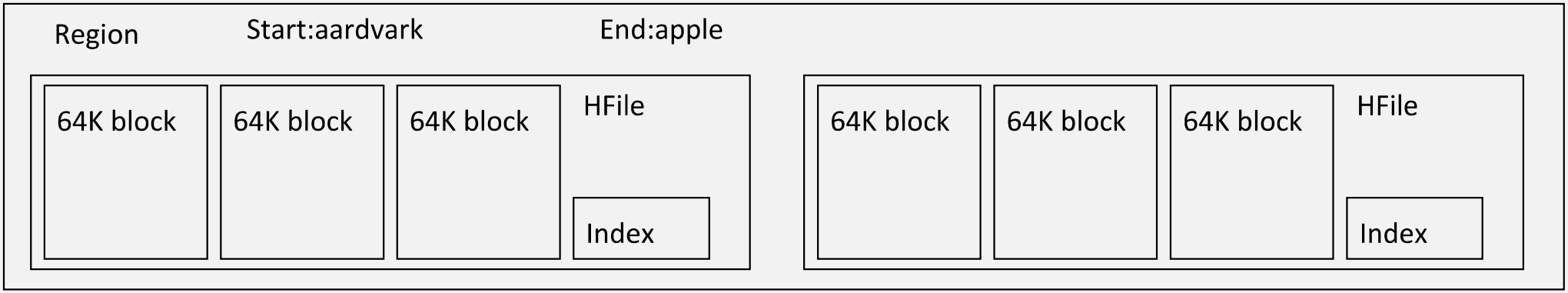
- Sequence of blocks on disk plus an index for block lookup
  - Can be completely mapped into memory
  - MemStore
- Supported operations:
  - Lookup value associated with key
  - Iterate key/value pairs within a key range



Source: Graphic from slides by Erik Paulson

HRegion

- Dynamically partitioned range of rows
- Built from multiple HFiles



Source: Graphic from slides by Erik Paulson

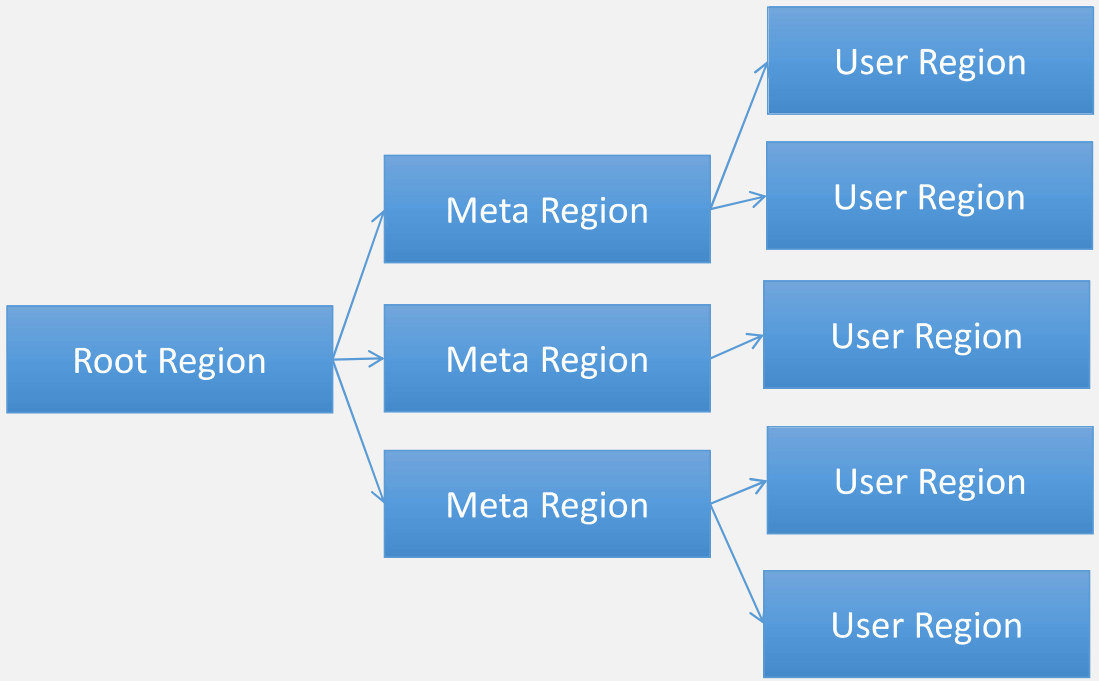
HBase Master

- 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

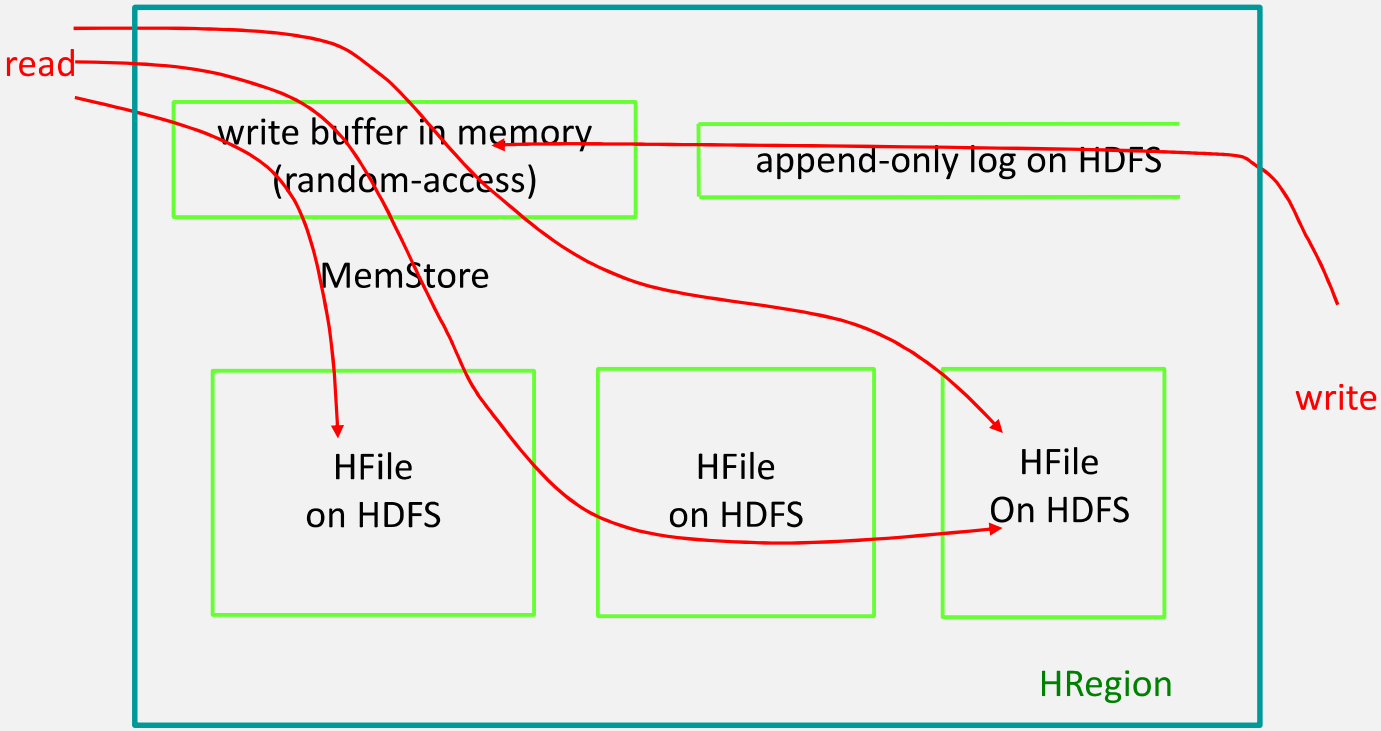
HBase Internals - Part 2

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

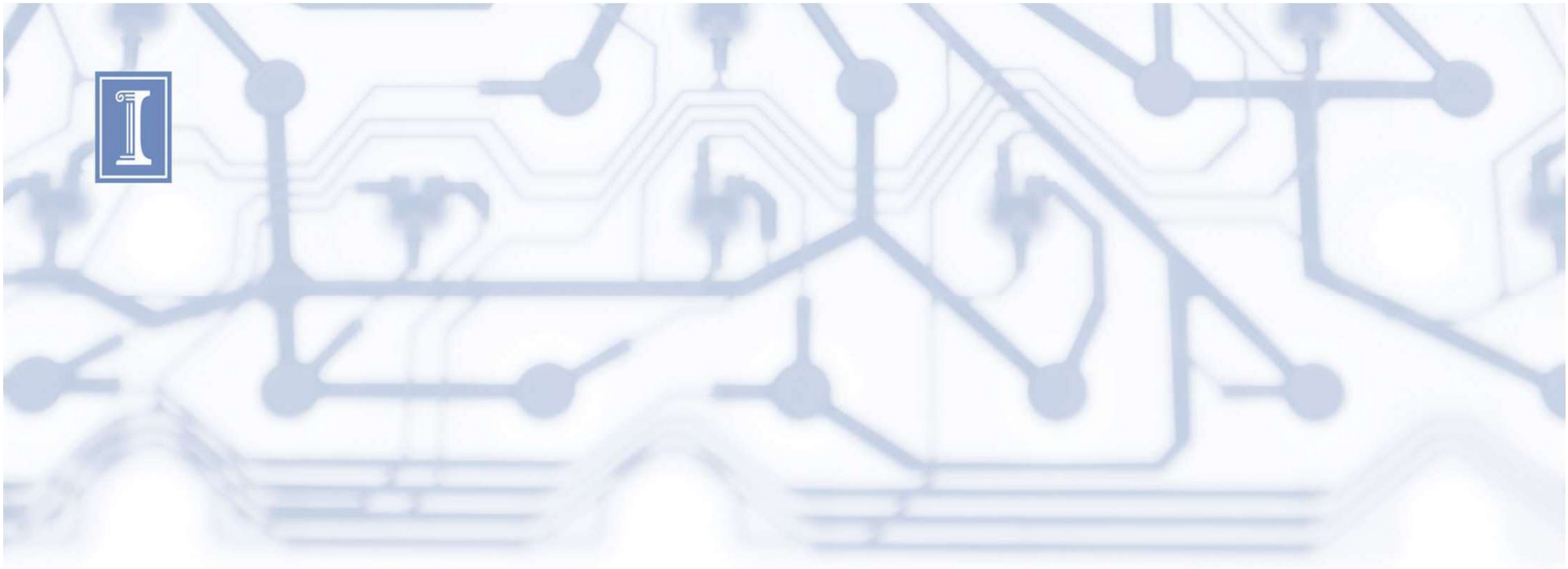
- Shell: allows interactive queries
- Driver: session handles, fetch, execute
- Compiler: parse, plan, optimize
- Execution engine: DAG of stages (MR, HDFS, metadata)
- Metastore: schema, location in HDFS, etc.

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

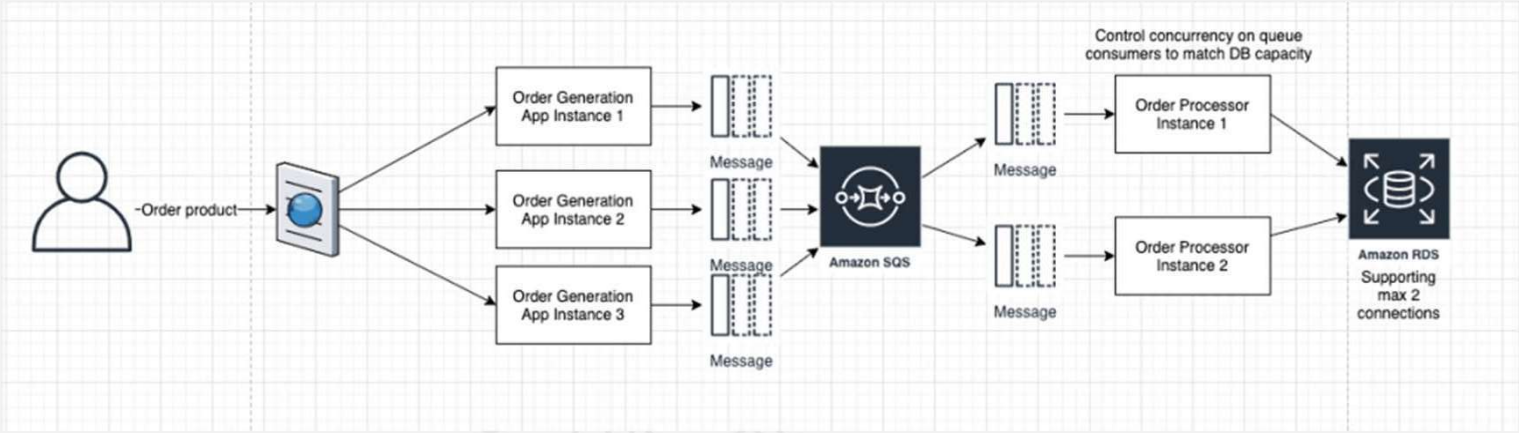
Decoupling

- 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



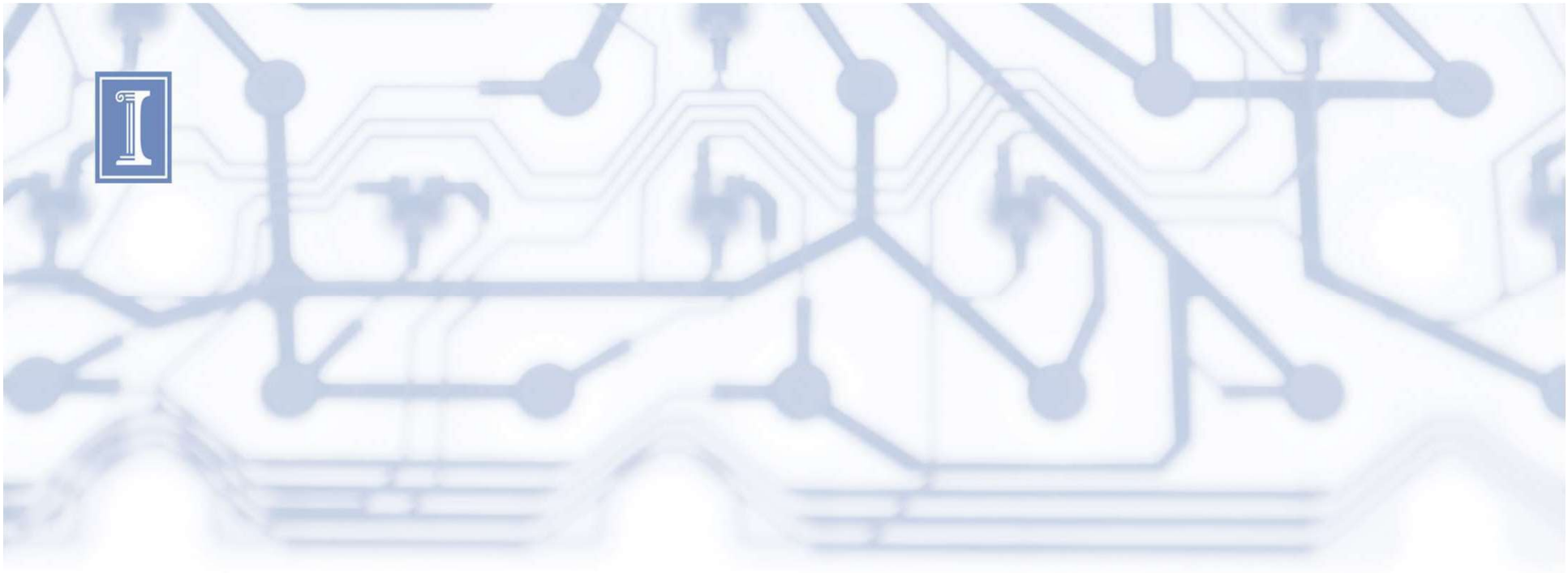
https://aws.amazon.com/blogs/architecture/application-integration-using-queues-and-messages/

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

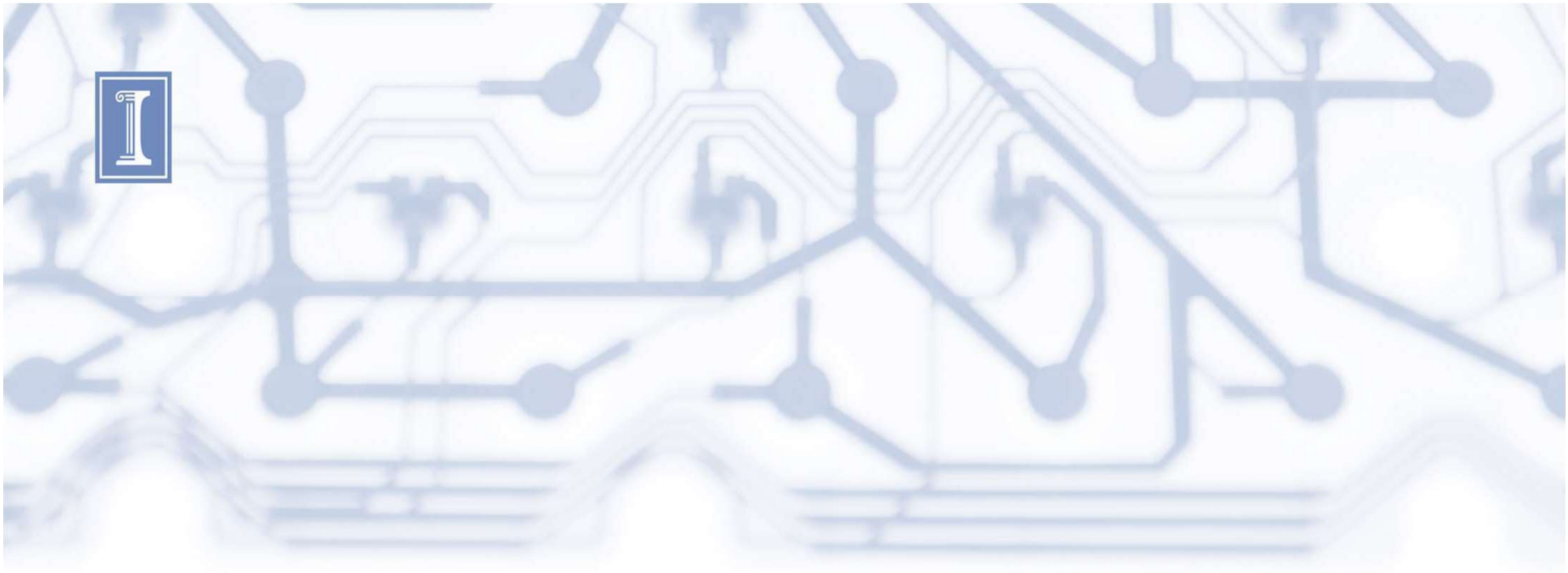
Recommended reading: https://sookocheff.com/post/messaging/dissecting-sqs-fifo-queues/

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



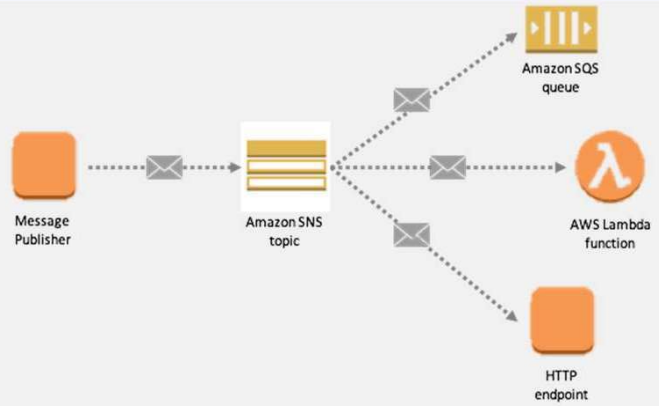
CLOUD COMPUTING APPLICATIONS

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

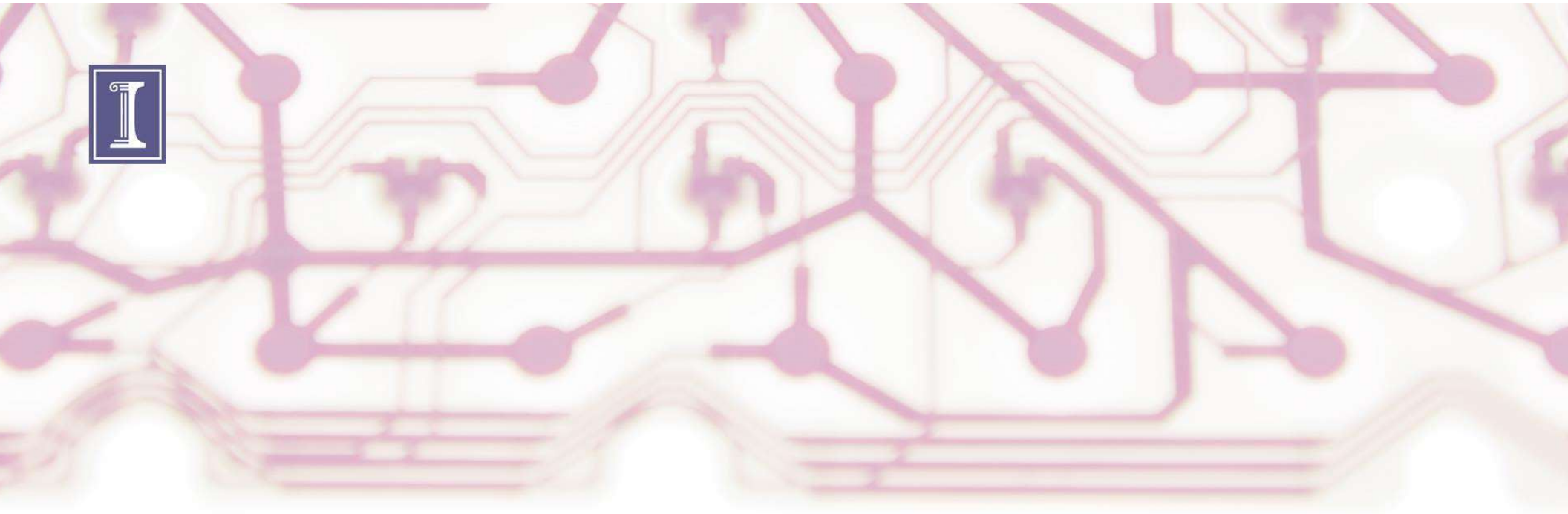
SNS Messages

- 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

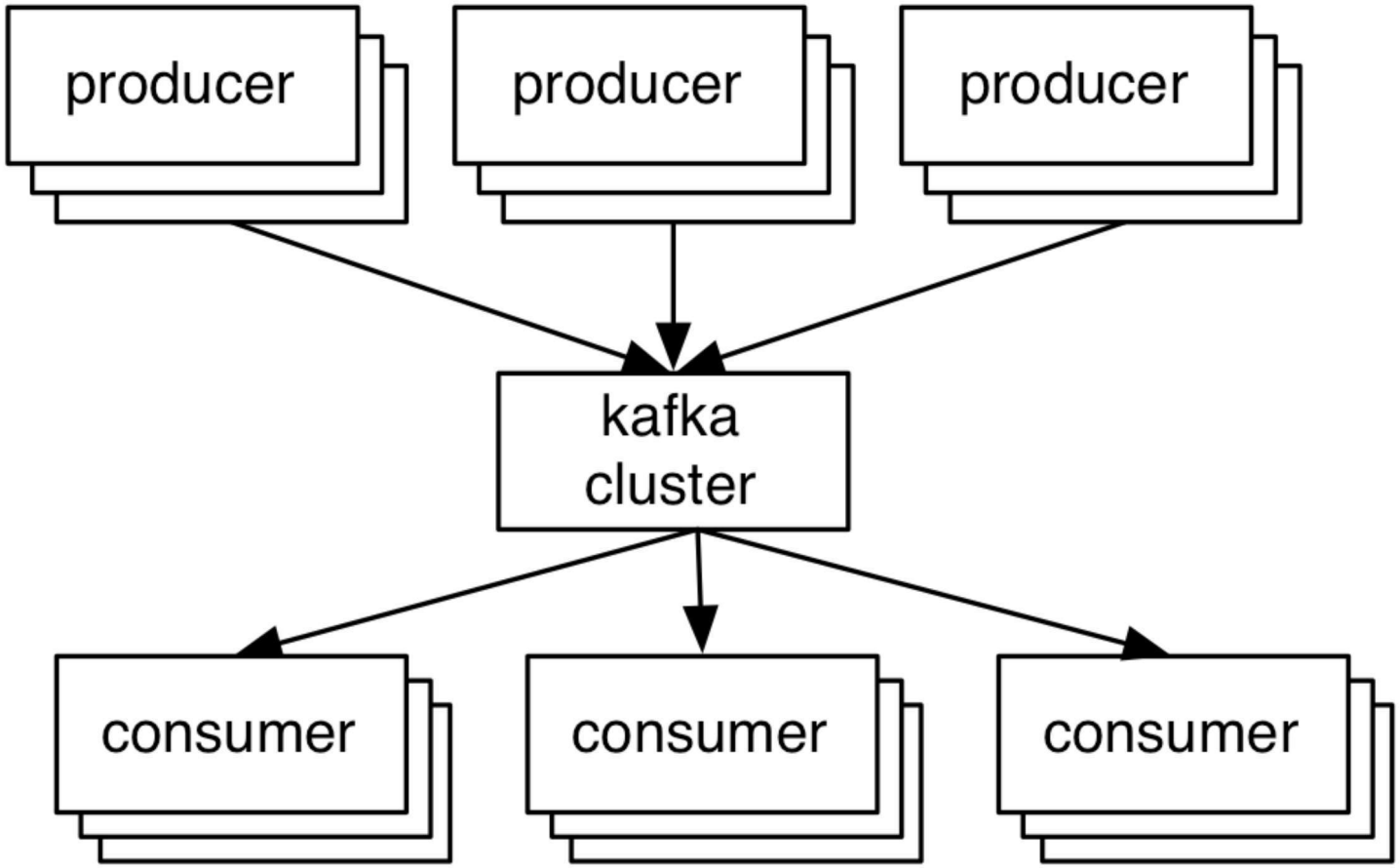
Roy Campbell & Reza Farivar

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



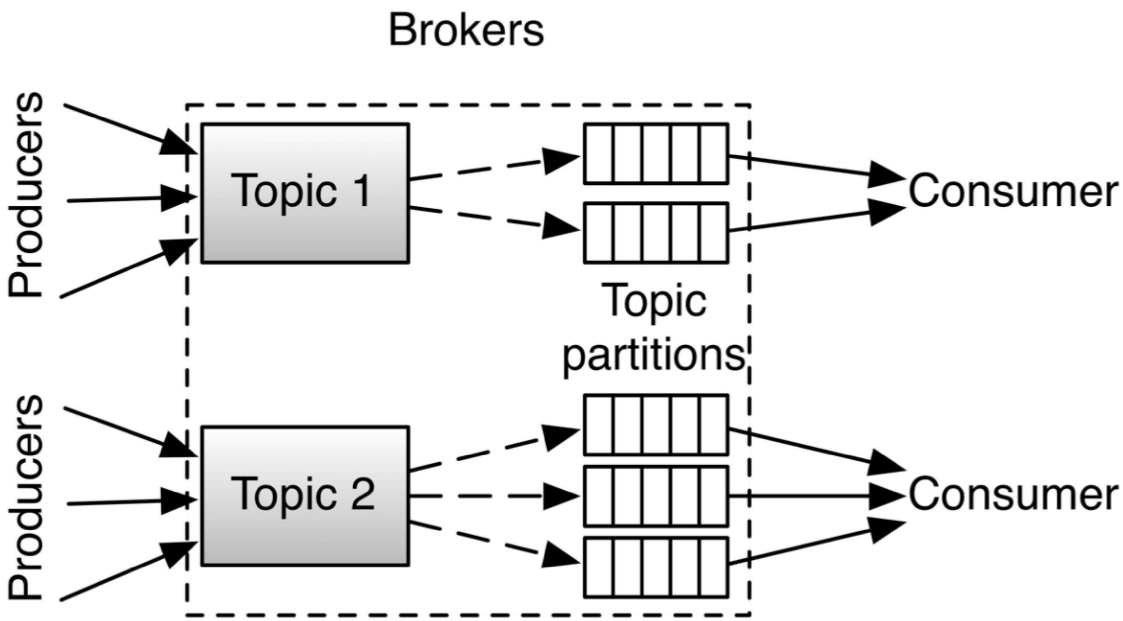
Description

- 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

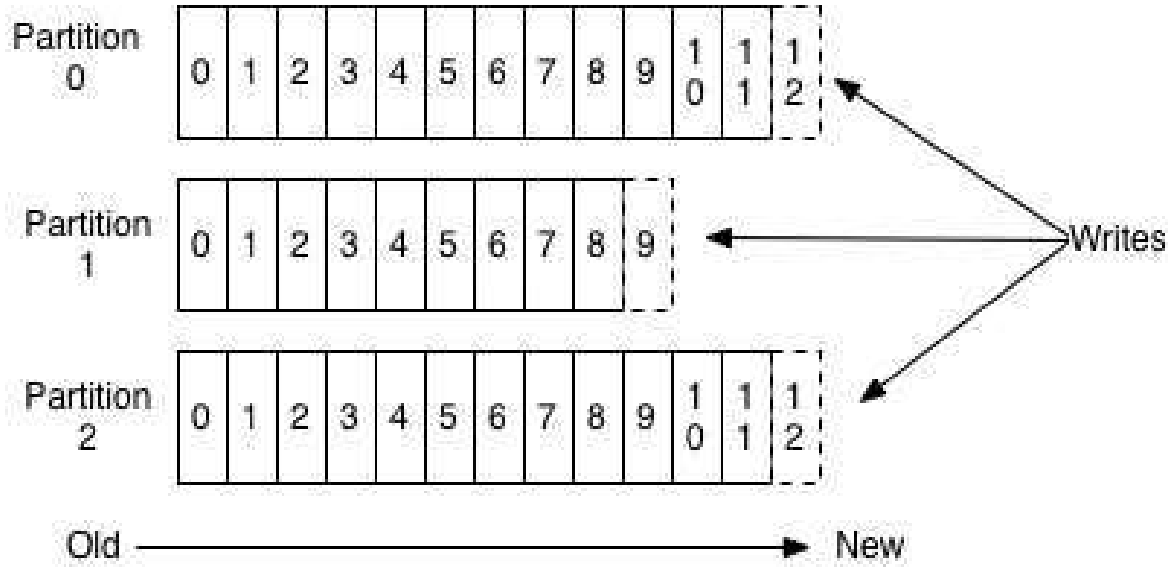
Topics



- 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.

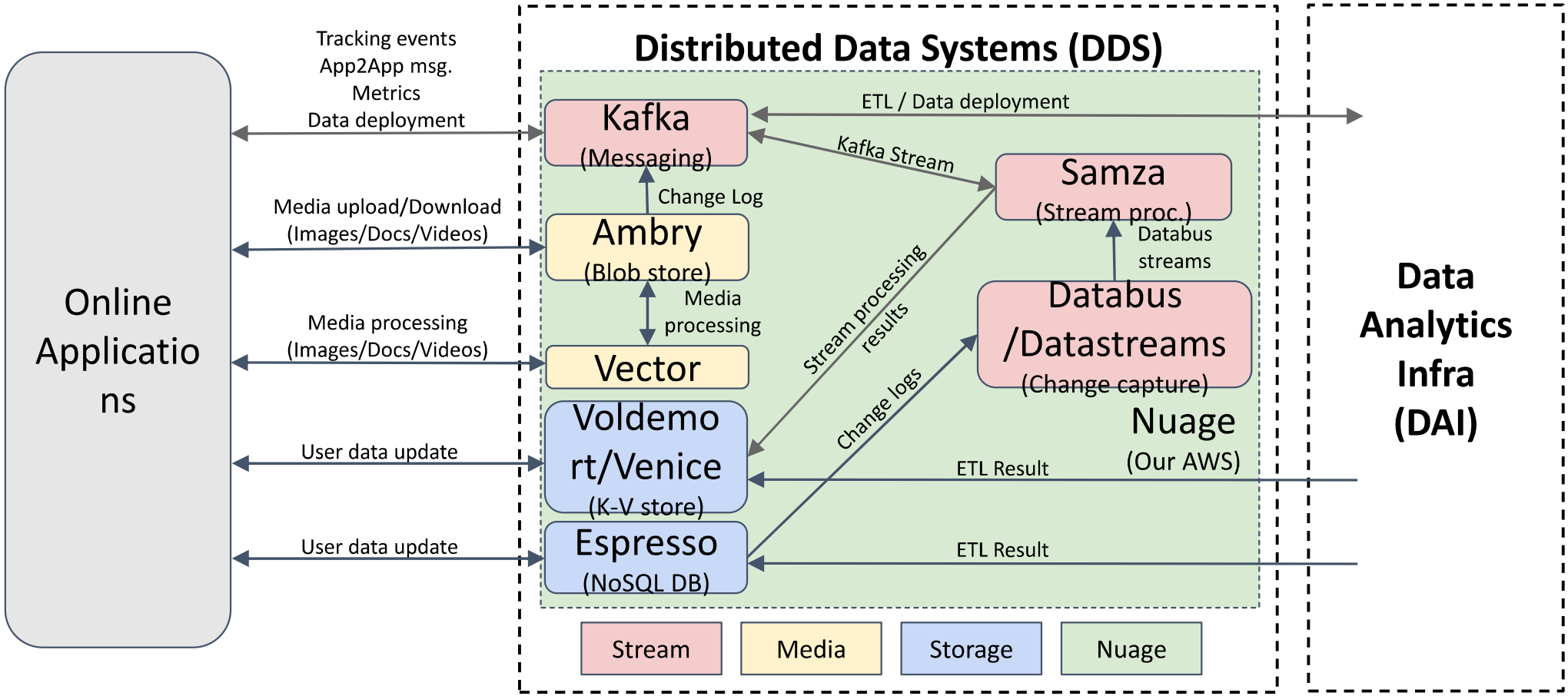


Each partition is an ordered, numbered, immutable append only sequence of messages--- like a commit log.

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

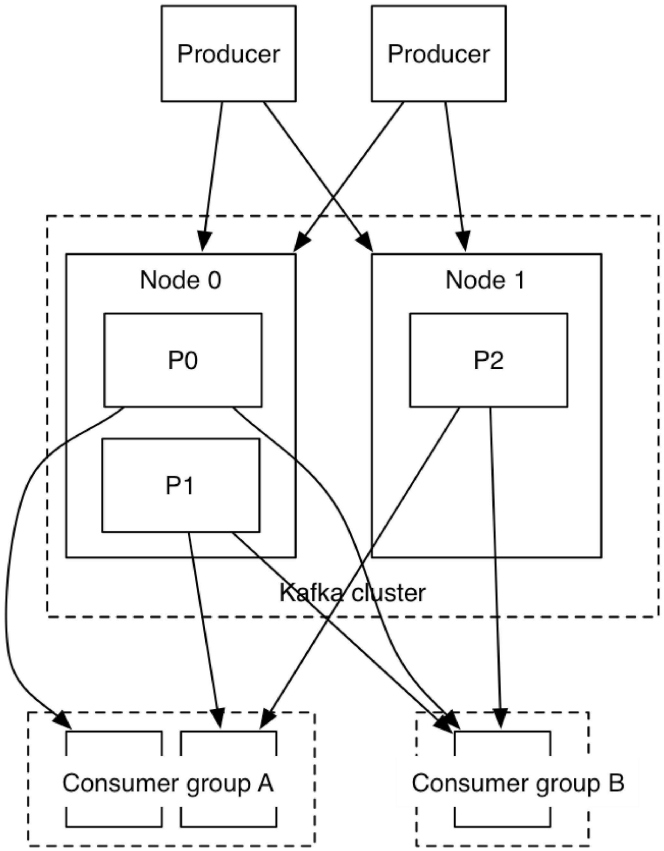
Kafka in a big picture (Linked In)



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

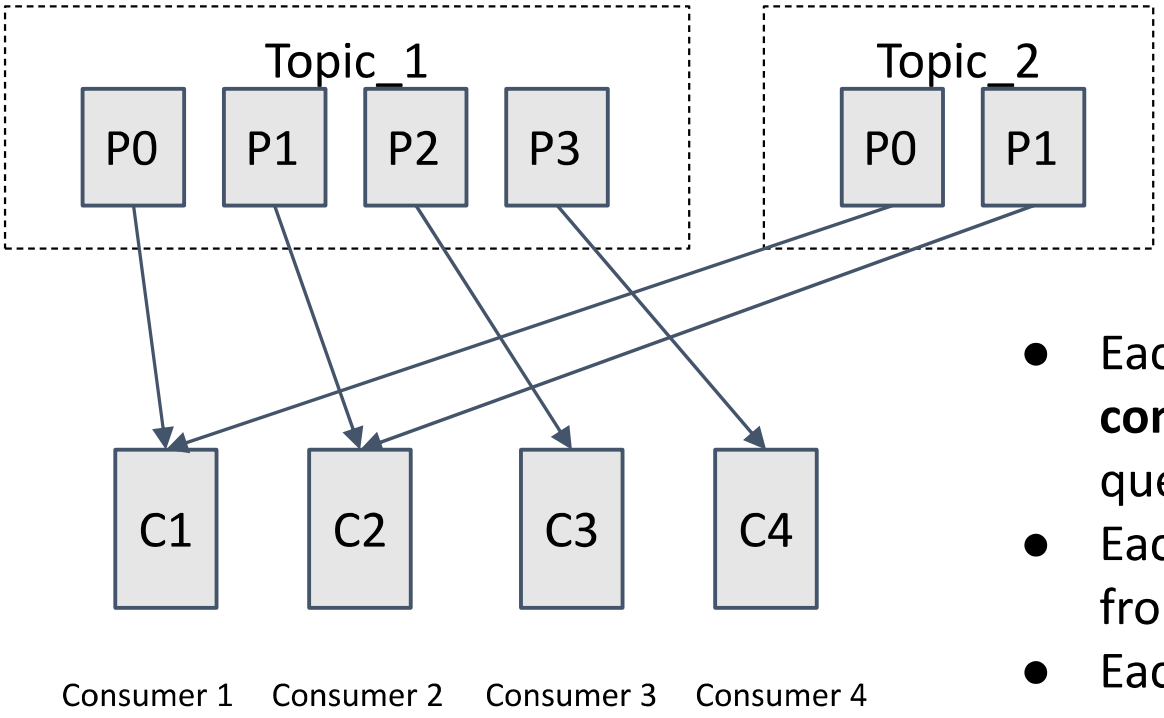
From Consumers' View

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

More about Consumer Offsets

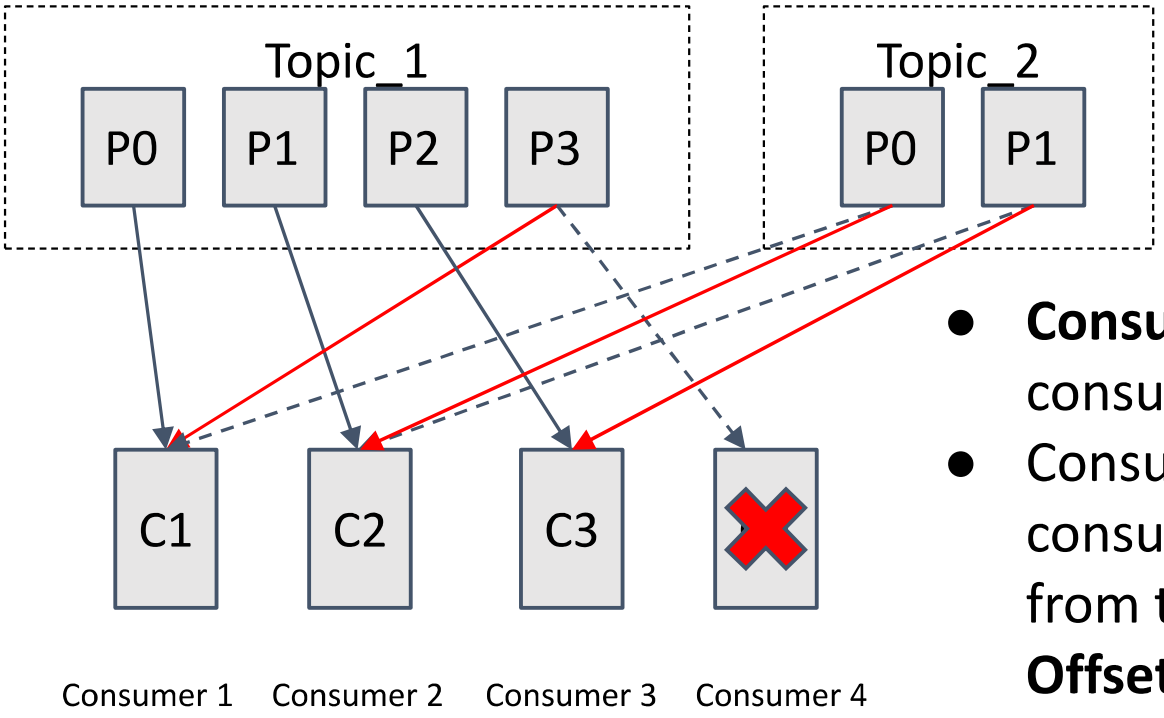
- Consumer Offsets are per
  Topic/Partition/ConsumerGroup
  (For a given group, look up the last consumed position in a topic/partition)
- Consumer Offsets can be **committed** as a checkpoint of consumption so it can be used when
  - Another consumer in the same CG takes over the partition
  - Resuming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