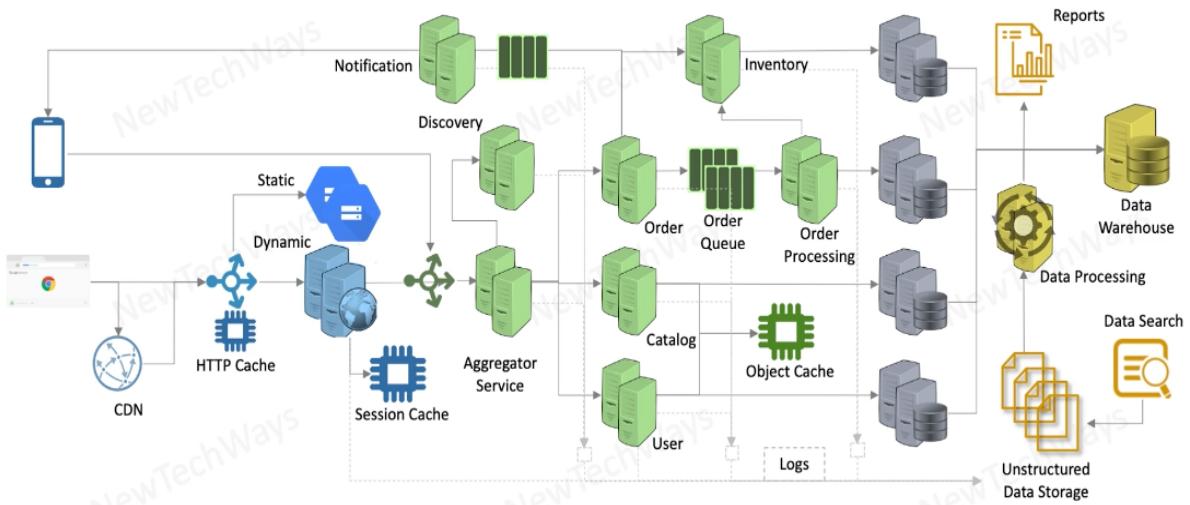


## System Design Tech Stack

Reference System:



Web application – Solutions:

Static Content →

- Apache Web Server
- Nginx Web Server
- Cloud Storage

Dynamic Content →

- Web Server – Apache HTTPD, NodeJS
- Java Container – Tomcat, Jetty, Spring Boot

Content Caching →

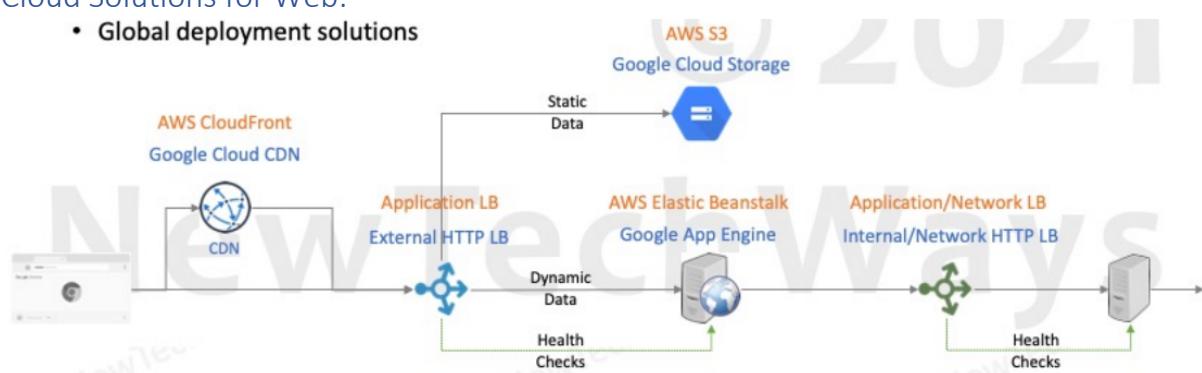
- Nginx

Content Distribution →

- CDN

Cloud Solutions for Web:

- Global deployment solutions



Load Balancer types based on content type:

- Static data – Cloud Storage, Dynamic Data – Web Application

- AWS – Application LB
- Google – External HTTP LB
- Level 7 LB can inspect content of the request.

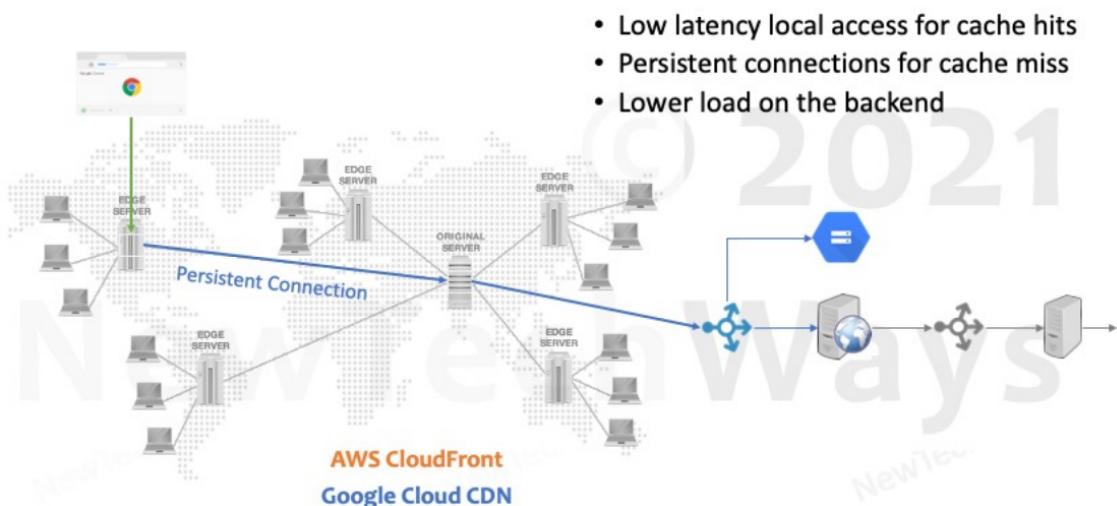
Load Balancer based on URL:

- AWS – Network LB
- Google – Network HTTP LB

If it's a Network LB CDN can directly talk to Cloud Storage

Cloud CDN →

## Cloud CDN



Services – Solutions:

Web Containers →

- Rest and Spring Containers

Object Caching →

- Memcache
- Redis

Async Messaging →

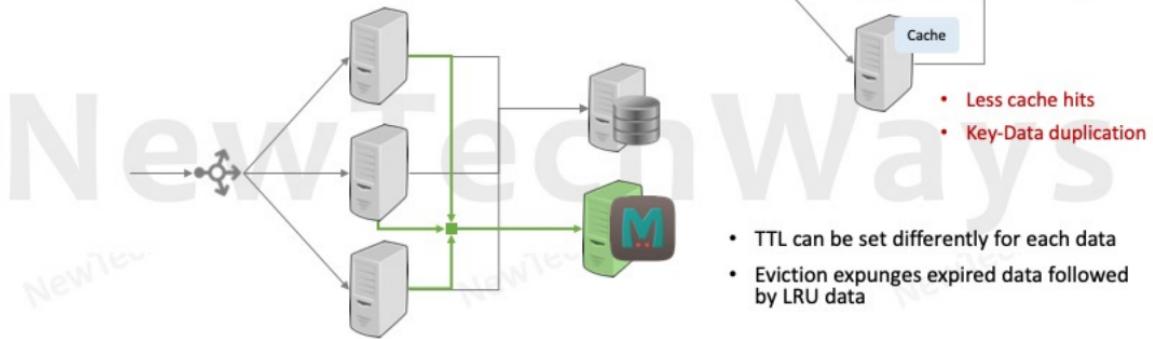
- Redis
- RabbitMQ
- Kafka

Object Caching →

Memcache:

## Memcached

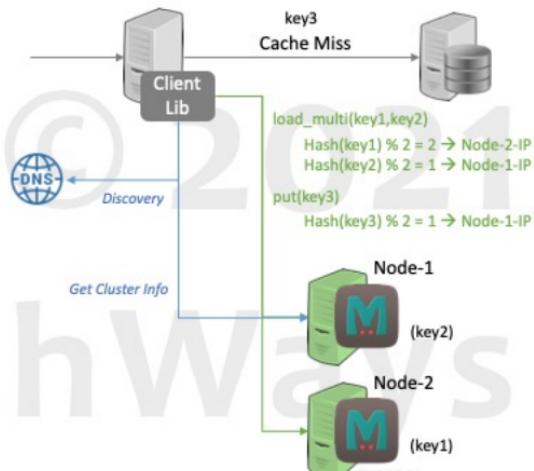
- Stores key value pairs
- Values can be
  - Any blob
  - Any size
    - Preferred < 1 MB
    - Max is configurable



- TTL can be set differently for each data
- Eviction expunges expired data followed by LRU data

## Memcached Architecture

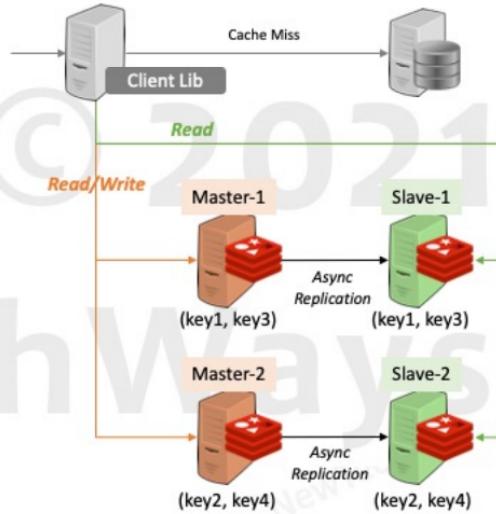
- Cache-aside pattern
  - Sub-millisecond Latency
  - Horizontally Scalable
    - Data is partitioned
  - High throughput
    - Parallel operations
  - Cluster aware client library
    - Consistent hashing for resolving a node
  - Node failure is treated as a cache miss
    - Use large number of nodes with less data
  - Data is lost if a node crashes or restarted
- 
- Client knows the IP address of all the nodes in a cluster. Client uses a memcache client lib. Client lib makes a DNS call to get any one of the Node info. Client makes a call to the Node to get the cluster info and in return gets the info of the entire cluster (how many nodes are there, IP address of all the nodes)
  - Cache-aside pattern – Client can only retrieve those value which client has put.
  - Client knows which Node has the value by **Hash(key) % N = Node number** ( $N$  – Number of nodes) or **Consistent hashing**.
  - Load multiple key into one API



*Redis Cache:*

## Redis Cache

- Much like Memcached
- It is a [ Key → Data structure ] store
  - Strings, Lists, SortedSets, Maps, ...
- **Data-Store mode for Persistence**
  - Stores data on a disk
  - Allows backups
  - Can be started pre-populated with a backup
- **Data Replication**
  - Asynchronous and synchronous
  - Read load distribution
  - High Availability
- Can also be used as a messaging queue
- **Data-Store mode requires fixed number of nodes**
  - Cache mode is suitable for node scaling



*Cloud Caching Solution:*

## Cloud Caching Solutions

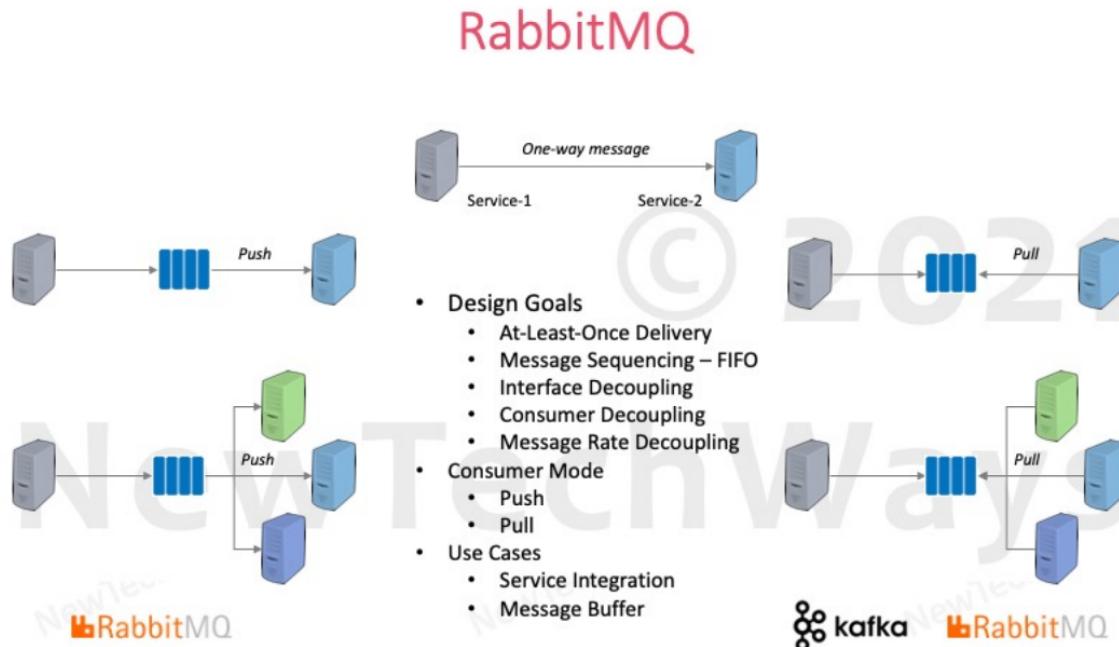
- AWS Elastic Cache
  - Memcached
  - Redis
- Google Memorystore
  - Memcached
  - Redis



- Fully Managed
- Sub-millisecond latency
- Scalable to 5 TB
- Highly available (99.9%)

## Async Messaging →

RabbitMQ:

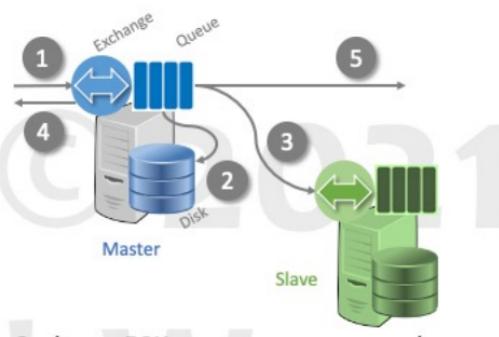


- RabbitMQ can be a push or a pull-based messaging whereas Kafka can only be a pull-based messaging.
- RabbitMQ is generally used as push-based messaging system.
- Service Integration - At times when there is message or there might not be message in the queue push-based messaging system is recommended (RabbitMQ) as in this case the consumer will unnecessarily poll and waste resources.
- Message Buffer - At times when the volume of message is very high (Streaming workflows) and queue is always full then pull-based messaging system is recommended (Kafka).

RabbitMQ Architecture:

### Rabbit MQ

- General purpose message broker
- Messages are pushed to consumers
- Message deleted once acknowledged
- Message ordering is guaranteed
- Useful for asynchronous service integration
- Both persistent and transient messages are supported
- Uses built-in component called exchange for routing



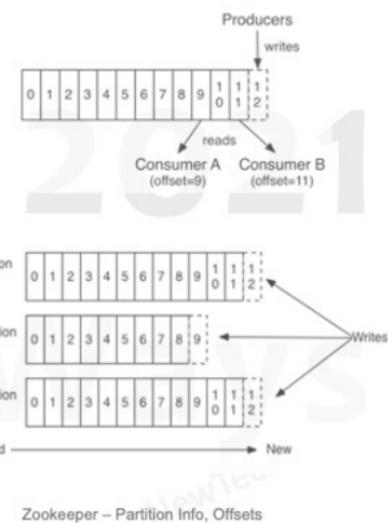
- Scales to 50K messages per second
- Scales well vertically
- Not horizontally scalable
  - Master-Slave replication for high availability
    - Clients can connect to any node
    - All publish, consume operations first go to master

- Transient messaging – when we need very high-speed processing of messages and we do not care if messages are lost. Messages not stored to DB.
- Persistent messaging – when we need guarantee that the message is at least delivered once. Messages stored to DB.

*Kafka Architecture:*

## Kafka

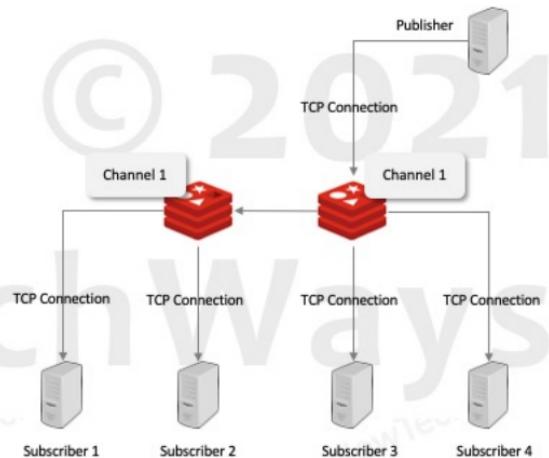
- Performance
  - Million messages per second
  - Sequential writes and reads on log files
  - Relies on page cache for quick reads
- Horizontally Scalable
  - Topics are partitioned
- Order of data guaranteed only within a partition
  - Producer can decide the partition
- Messages are not deleted, so can be replayed
- Consumers can only pull the data
  - No push by Kafka
  - Not designed for service integration
- Useful for streaming analytical workloads
  - Where high throughput is required
  - Click streams, Page views, Logging, Ingestion, Security



*Redis Pub/Sub:*

## Redis Pub/Sub

- For short lived messages with no persistence
  - Much like a synchronous call
  - Fire and forget
    - No delivery guarantee
- Millions operations per second
- Useful for making dashboards
  - Leaderboard
- Comparison
  - Kafka
    - No push
    - Writes to log
  - RabbitMQ Transient
    - Delivery acknowledgement
    - Deletion of delivered messages
- Fast messaging without any persistence, we should consider Redis.
- In Redis topics are called as Channel.



Cloud MQ Solutions:

## Cloud MQ Solutions

Community	AWS	Google Cloud
Rabbit MQ	SQS	Pub/Sub
Kafka	Kinesis	

Datastore – Solutions:

RDBMS →

- Oracle
- SQL Server

Distributed Databases →

- Key-Value – Dynamo
- Column Family – Big Table, Cassandra HBase
- Document Oriented – MongoDB

RDBMS →

## RDBMS

- General purpose database (< 1-5 TB data, 10K connections)

- ACID Transactions

- Update of multiple records or tables

Possible only on a single node or else it requires 2PC/3PC

- Data Consistency

- Data same for all readers at any given time

Leads to low availability

- Fixed Schema

- Data analysis

Impedes application evolution

- Columns can be selected

- Rows can be filtered

- Queries can Join Tables

- Query pattern can change or evolve

- Any column can be indexed

- Indexes can be created whenever required

Joins may slow down a system

- Normalized Data

- Efficient storage and writes

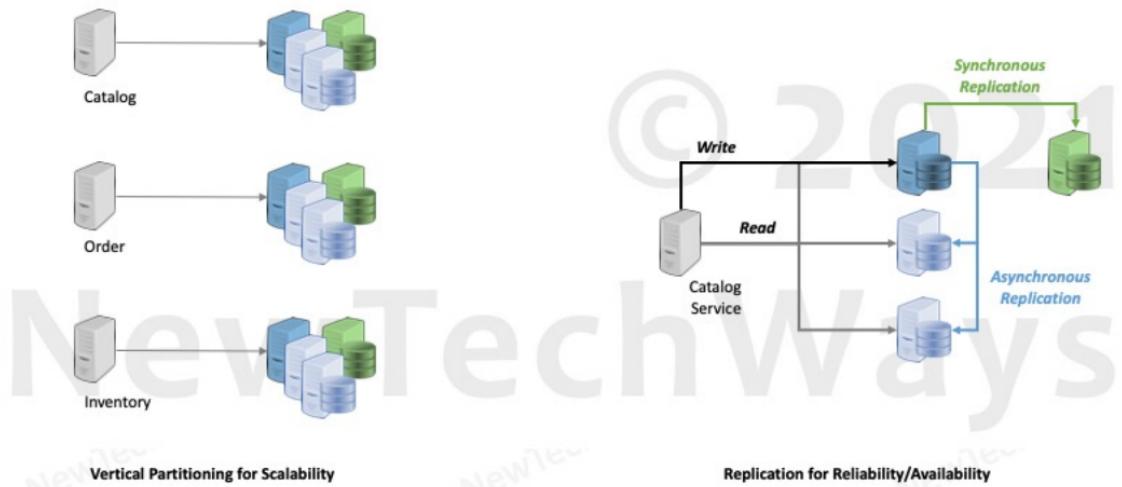
Allows only one version of data

- Overwrite for updates

Old design for expensive disk space

RDBMS Scalability Architecture:

## RDBMS Scalability Architecture



- Separate DB for each service.
- Read replicas (Async data replication).

Distributed Databases →

Dynamo DB:

## Amazon DynamoDB

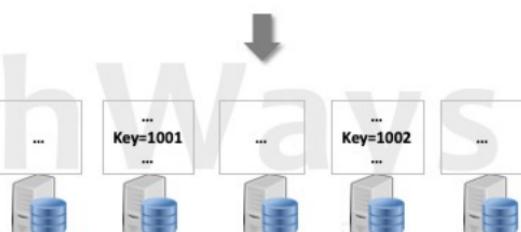
- Key-Value Pair Datastore
- Used for High Scalability & Availability
- Table is a Hash-Map
  - Persistent
  - Distributed
- API
  - Put, Get, Update, Delete, Query
- Index Key – has two parts
  - Partition Key
    - Can have only one attribute
    - Key is hashed to determine the partition
  - Sort Key
    - Determines sort order of an item within a partition
    - Can be used for range query  $<$ ,  $>$ , like
- R/W ops for a key are atomic
  - Table as HashMap

Primary Key

Value

Partition Key      Sort Key

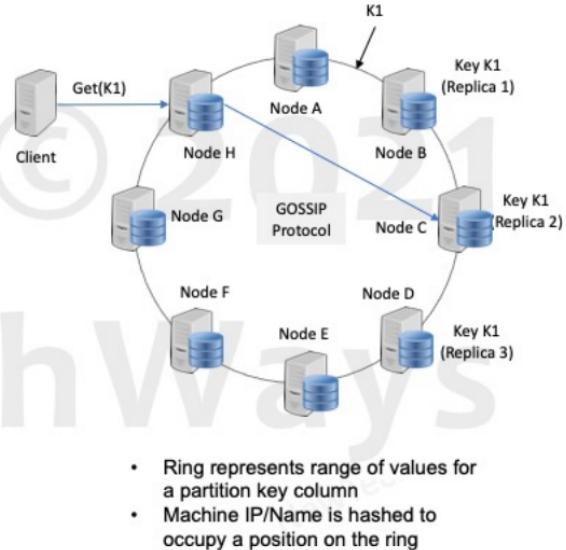
Product Id	Size			
1001	L	name="Blue Denim"	qty=150	
1001	XL	name="Blue Denim"	qty=780	
1002	M	name="Black Shirt"	qty=960	discount=10
1002	L	name="Black Shirt"	qty=300	discount=20



Dynamo DB architecture:

## Dynamo DB Architecture

- Suitable for storing small chunks of data
  - Key values less than 1MB
- Peer-To-Peer cluster
  - No master – write on any node
  - Each node responsible for a set of keys
  - Consistent hashing of virtual nodes for key allocation
- Highly Scalable
  - Practically any number of nodes
  - Petabytes of data
  - Can handle 10 million RW ops requests/second
- Highly Available
  - Updates are not rejected even in case of network partitions or server failures
  - Vector clocks and business rules to resolve merge conflicts
- Consistency guarantee is adjustable
  - $R+W > N$  for strong consistency



Google Big Table:

## Google Big Table

- Basis for Apache HBase
- Column-Family Storage
- Table as Tree-Map
  - Sparse
  - Sorted
  - Persistent
  - Distributed
- Limited CF (< 100)
- CFs are compressed
- Unlimited columns
- R/W ops atomic for a key
- Timestamps for versioning

Sorted

Row Key	Document:	Language:	Referrer: trips.com	Referrer: holidays.com	Referrer: news.com
com.airlines.www	<!DOCTYPE html><html>..	EN	Check Flights		
com.hotel.www	<!DOCTYPE html><html>..	EN		Your Hotel	Venue

Sparse

Index Key: Row Key:Col Family:Col Name/timestamp

Row Key	Column Family	Column	Timestamp	Value
com.airlines.www	Document		987654321	<!DOCTYPE html><html>..
com.airlines.www	Language		987654321	EN
com.airlines.www	Referrer	trips.com	987654321	Check Flights
com.hotel.www	Document		987655432	<!DOCTYPE html><html>..
com.hotel.www	Language		987655432	EN
com.hotel.www	Referrer	holidays.com	987655432	Your Hotel
com.hotel.www	Referrer	news.com	987655432	Venue

- Apache HBase is open-source implementation of Bigtable.
- Table as TreeMap
- If a new value needs to be updated, then the old value is also kept and a new row is added for the new value.
- Rows are in a sorted order.

## *Bigtable Architecture:*

# Bigtable Architecture

- Schema-less, structured (CF), sorted data
  - GFS for reliable persistent storage
    - Replicated storage for large files
  - In-memory tablet servers
    - High R/W throughput with low latency
    - Single copy of data for read and write
      - No write conflicts
  - Horizontally scalable
    - Stores peta/exabytes of data
    - Client load is distributed
  - Strongly consistent

Host Name	Document Library	Location	Owner	Notes
contoso.sharepoint.com	Language	http://sharepoint/	SP2010Admin	
contoso.sharepoint.com	InfoPath	http://sharepoint/	Share Rights	
contoso.sharepoint.com	Language	http://sharepoint/	SP2010Admin	(Check Rights)
contoso.sharepoint.com	InfoPath	http://sharepoint/	Share Rights	
contoso.sharepoint.com	Language	http://sharepoint/	SP2010Admin	Year Hotel
contoso.sharepoint.com	InfoPath	http://sharepoint/	Share Rights	
contoso.sharepoint.com	Language	http://sharepoint/	SP2010Admin	
contoso.sharepoint.com	InfoPath	http://sharepoint/	Share Rights	
contoso.sharepoint.com	Language	http://sharepoint/	SP2010Admin	
contoso.sharepoint.com	InfoPath	http://sharepoint/	Share Rights	
contoso.sharepoint.com	Language	http://sharepoint/	SP2010Admin	
contoso.sharepoint.com	InfoPath	http://sharepoint/	Share Rights	

*Data ranges are split into tablets*

- The diagram illustrates the GFS architecture across three main layers:

  - Persistence Layer:** Contains multiple SSTable components.
  - In Memory Layer:** Contains multiple Tablet components.
  - R/W Operations layer:** Contains a Client Library, a Master Server, and a Lock Server (Chubby).

Key components and interactions:

  - Client:** Interacts with the **Client Library**.
  - Client Library:** Handles **R/W Operations** and interacts with the **Master Server**.
  - Master Server:** Manages **Tablet metadata** and interacts with the **Lock Server (Chubby)** to manage **Liveness Lock**.
  - Lock Server (Chubby):** Manages **Liveness Lock** for the tablet servers.
  - Tablet Servers:** Manage **Bulk R/W** operations and interact with the **Tablet** component in the In Memory Layer.
  - Tablet:** Manages **Bulk R/W** operations and interacts with the **SSTable** component in the Persistence Layer.
  - SSTable:** Provides persistent storage for data.
  - GFS (Colossus):** Manages multiple SSTable components.
  - Persistence Layer:** Manages multiple SSTable components.

**Annotations:**

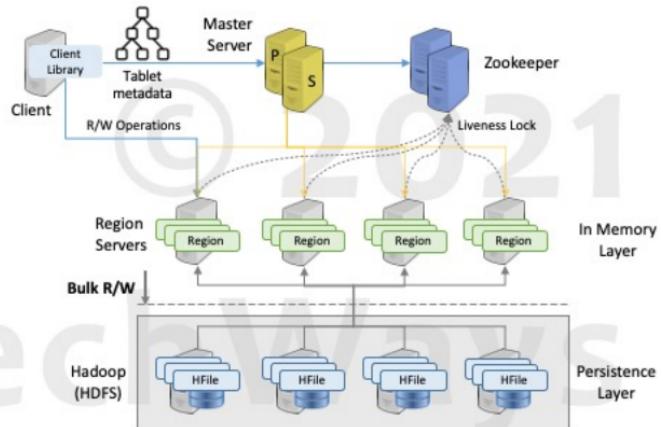
  - Master:** Caches tablet metadata and directly works with tablet servers.
  - Master:** Checks if tablet servers are live and assigns tablets to tablet servers.
  - GFS for persistent & reliable storage of log files and data files**

- Data is stored in sorted order that helps in range queries.
  - Bigtable uses GFS (Google File System) underneath as the persistent storage.
  - Hadoop is the open-source implementation of GFS.
  - Tablet Servers store the String Sorted table (SSTable) or GFS data in memory for faster access (as these files are very huge and stored in GFS disk and reading data will take time). It acts as a cache.
  - **Read** – During a read operation data is served from the Tablet Servers to the client.
  - If the data is not present, then it is first loaded into the Tablet servers from the GFS and then sent to client.
  - **Write** – During a write operation first data is written to a log (WAL – Write Ahead Log) then to the Tablet server and then to the SSTable (Write through Cache).
  - Client never connects to master server or Lock Server for any read write operations.
  - Client connects to the master server and lock server only once to get the Tablet Metadata which is a BTree kind of index which it will store in memory and that guide the client which Tablet to connect for Read and Write data.
  - Master role is to check if the nodes are alive and to assign tablets to nodes servers and it re-assigns the tablets if the node is not alive.
  - Every Tablet server is supposed to hold a lock in the lock server and if it fails to do so then it is considered as dead. Master also considers that server as dead and redistributes the data to other nodes.

*HBase:*

## HBase

- Open source implementation of Bigtable
- Column Family schema
- Keys are Range Partitioned
- Storage on Hadoop HDFS
- Strong consistency over high availability
- Highly scalable
- High throughput and low latency writes



- Tablet Servers (BigTable) → Region Servers (HBase)
- Underlying storage GFS (BigTable) → Hadoop HDFS (HBase)
- Lock Server Chubby (BigTable) → Zookeeper (HBase)

*Cassandra:*

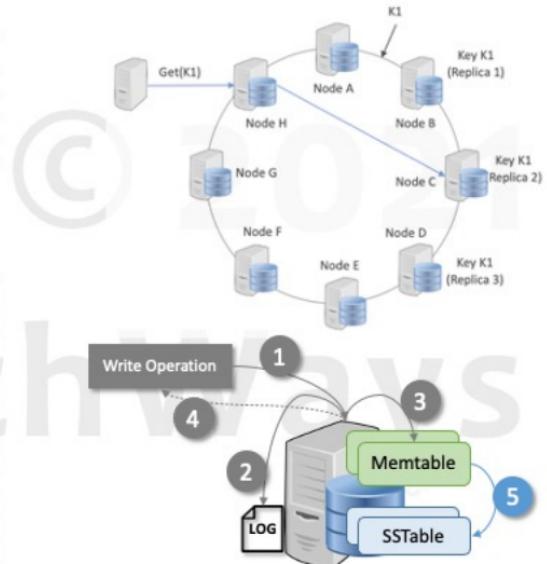
## Cassandra

Partition Key	Sort Key	Columns		
Product Id	Size	Name	Quantity	Discount
1001	L	Blue Denim	150	
1001	XL	Blue Denim	780	
1002	M	Black Shirt	960	10
1002	L	Black Shirt	300	20

Column Family

Partition X	Product Id	Size	Col Name	Col Value
Partition X	1001	L	Name	Blue Denim
	1001	L	Quantity	150
Partition Y	1001	XL	Name	Blue Denim
	1001	XL	Quantity	780
	1002	M	Name	Black Shirt
	1002	M	Quantity	960
Partition Y	1002	M	Discount	10
	1002	L	Name	Black Shirt
	1002	L	Quantity	300
	1002	L	Discount	20



- In Cassandra the sorting is only in a partition.
- Data is sent to a partition based on the hash of the partition key.
- Data model is mix of DynamoDB and BigTable.
- Peer to Peer like DynamoDB.
- Write operation is similar to BigTable → 1. log file (WAL), 2. Memtable ,3. SSTable
- Tablet Server (Big Table) → Memtable (Cassandra)
- Read operation is similar to BigTable.

*MongoDB:*

# MongoDB

- Key -> Document
  - In Binary JSON (bson) format
- Columns created dynamically
- Columns can be indexed
- Documents can be queried
  - on id and or column values
- An operation on a single document is atomic
  - 2 PC for multiple documents

```

db.inventory.insertMany([
  { item: "journal", qty: 25, size: { h: 14, w: 21, uom: "cm" }, status: "A" },
  { item: "planner", qty: 75, size: { h: 22.85, w: 30, uom: "cm" }, status: "D" },
  { item: "postcard", qty: 45, size: { h: 10, w: 15.25, uom: "cm" }, status: "A" }
]);

db.inventory.find( { status: "A", qty: { $lt: 30 } } )   Insert

db.inventory.find( { "size.h": { $lt: 15 } } )   Query

db.inventory.updateOne(
  { item: "paper" },
  {
    $set: { "size.uom": "cm", status: "P" },
    $currentDate: { lastModified: true }
  }
)   Update

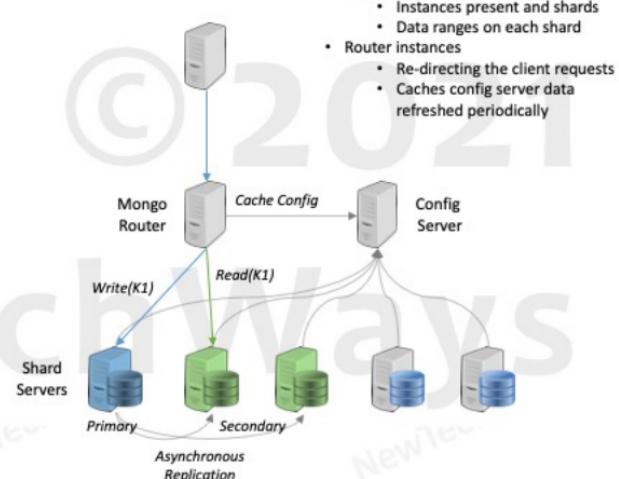
db.products.createIndex(
  { item: 1, quantity: -1 }   Index Creation
)

```

*MongoDB Architecture:*

# MongoDB Architecture

- Indexing for fast search
  - It's a write overhead
- Sharding for Scalability
  - Range sharding
  - Hash sharding
- Replication
  - Master Slave
    - No write conflicts
    - Asynchronous (default)
      - Eventual consistency
    - Synchronous (on demand)
- Works very well with Node.js
  - Javascript -> Node.js -> Mongo
    - JSON format
- In terms of latency, it is like RDBMS. The more the number of indexes the slower the write operations.
- MongoDB differs from RDBMS in terms of horizontal scalability.



*Analytics:*

*Analytics – Solutions:*

*Data Movement →*

- Logstash

- Fluentd

Storage →

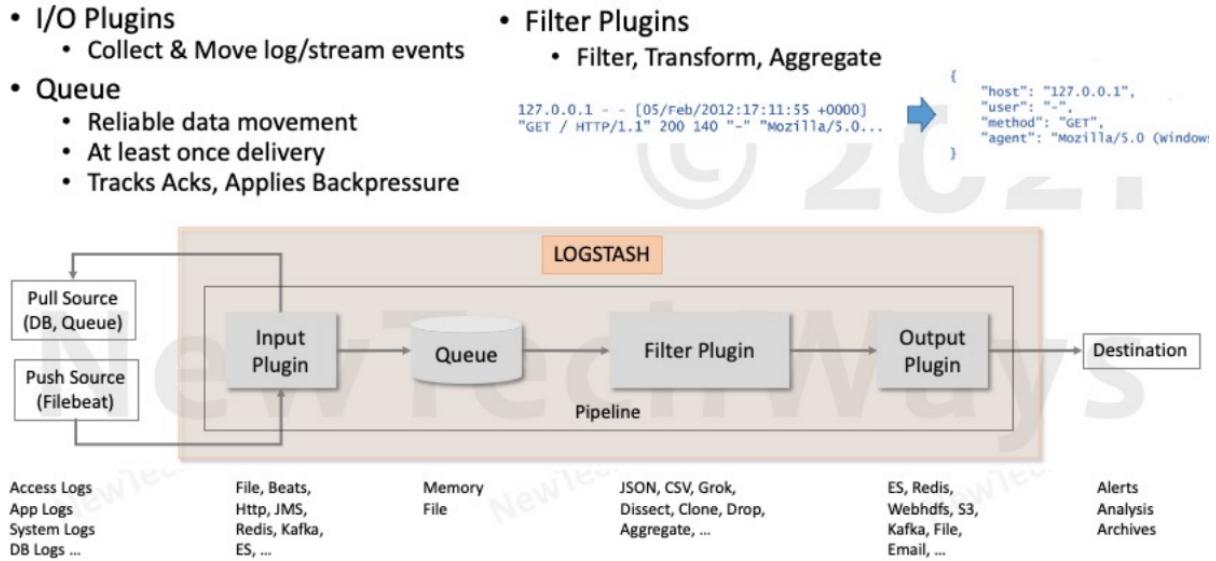
- Hadoop HDFS – MapReduce
- Elastic Search – Search

Stream processing →

- Kafka – Buffer
- Storm, Flink – Processing

*Logstash:*

## Logstash Architecture

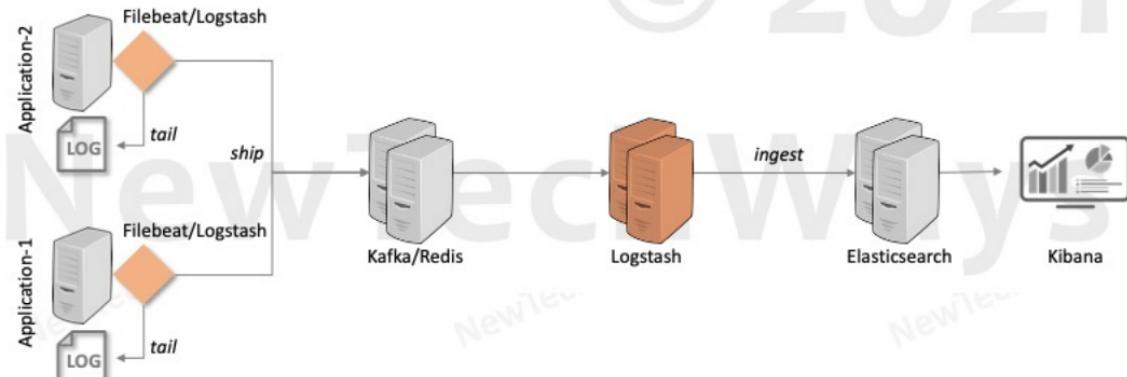


- Move files from services to storage.
- Filtering of data can also be done before storage.

*Logstash Data streaming architecture:*

## Logstash Data Streaming Architecture

- Streaming log data for Real-Time Analytics
- Horizontally Scalable & Highly Available – Any number of Logstash nodes
- Fault Tolerance – Needs reliable disk storage (RAID, Cloud persistent disks)
- Use Kafka as buffer – For heavy load that Logstash Queue cannot handle

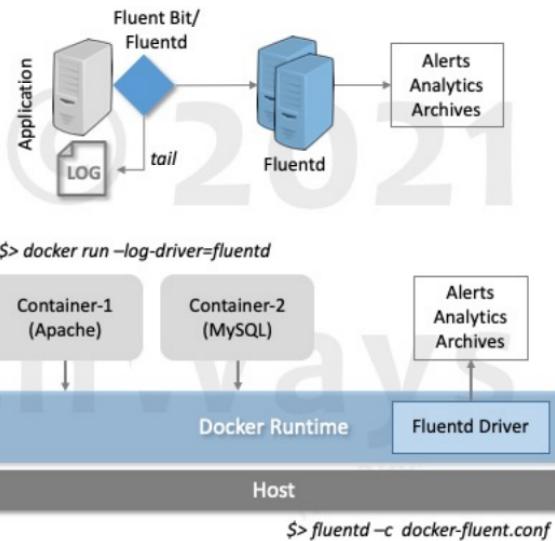


- Logstash is a heavy application, so it's not recommended to install on the same machine where the app is installed. Filebeat is another tool developed by Logstash team which is very light weight, and it can collect the data and ship it.
- FileBeat can send data directly to analytic engine but that won't be a scalable solution as when more number of application generate log the ES might not be able to handle.

*Fluentd: (d – demon)*

## Fluentd

- Older than Logstash
- Memory Footprint
  - Logstash – Heavyweight (GB)
  - Filebeat – Lightweight (MB)
  - Fluentd – Lightweight (MB)
  - Fluent Bit – Super Lightweight (KB)
- All features of Logstash
- + Routing
  - Tags
- + Docker Logging
  - Picks log events from container console



\$> docker run -log-driver=fluentd

Container-1  
(Apache)

Container-2  
(MySQL)

Alerts  
Analytics  
Archives

Docker Runtime

Host

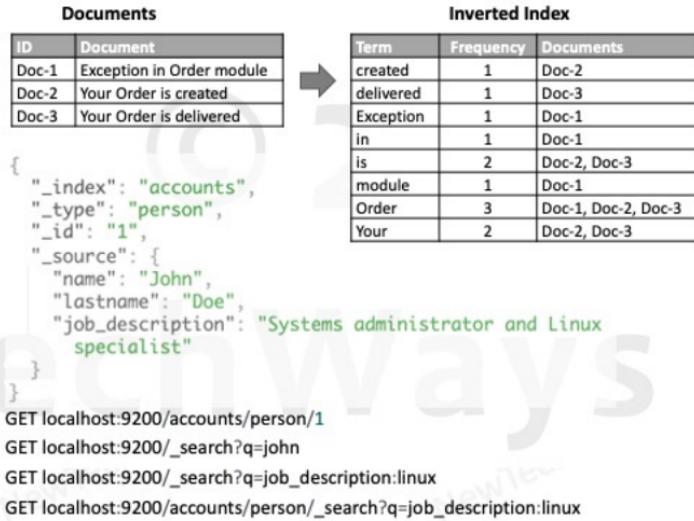
Fluentd Driver

\$> fluentd -c docker-fluent.conf

Elasticsearch:

# Elasticsearch

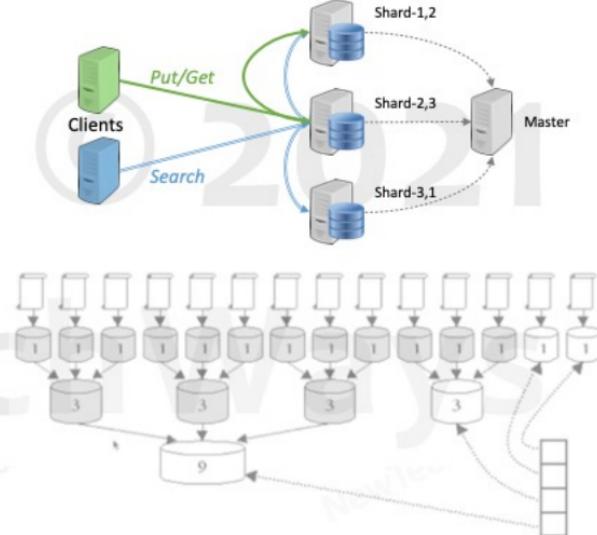
- Full-Text Search
  - Filter, Group, Aggregate
- Stores JSON Documents
- Document Fetched using id
- Indexes JSON keys and values
- Structure
  - Index -> Database
  - Type -> Table
  - Document -> Row
    - JSON keys are flattened
  - Supports data types
- Users can specify mapping between terms & documents



Elasticsearch architecture:

# Elasticsearch Architecture

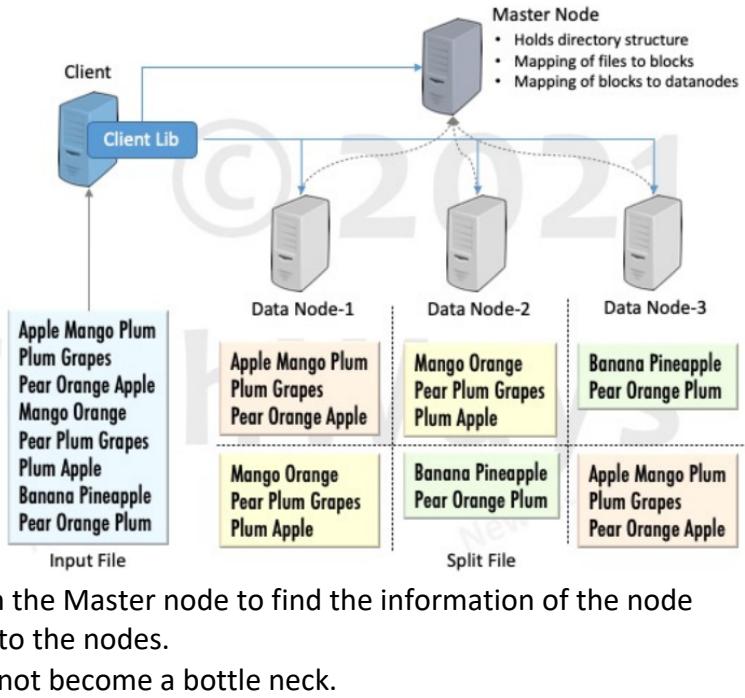
- Document Oriented data-model
- Horizontally Scalable
  - Petabytes of data
  - Data is sharded with key as Document Id
  - Put/Get request goes to specific shard
  - Search queries go to all shards
- Highly Available
  - Shards are replicated
- Index structure is based on merge sort
- Index not updated with every update or insert
- Index maintained in memory
  - Occasionally flushed to disk



Hadoop HDFS:

## Hadoop HDFS

- Distributed File Data Storage**
  - Unstructured data files
  - Petabytes of data
  - Large file size > 100MB
- Distributed files**
  - Files broken into chunks
  - For parallel reads
    - Map-Reduce
- Sequential Writes – Append**
  - Large blocks of data – 64MB
- Replication for reliability**

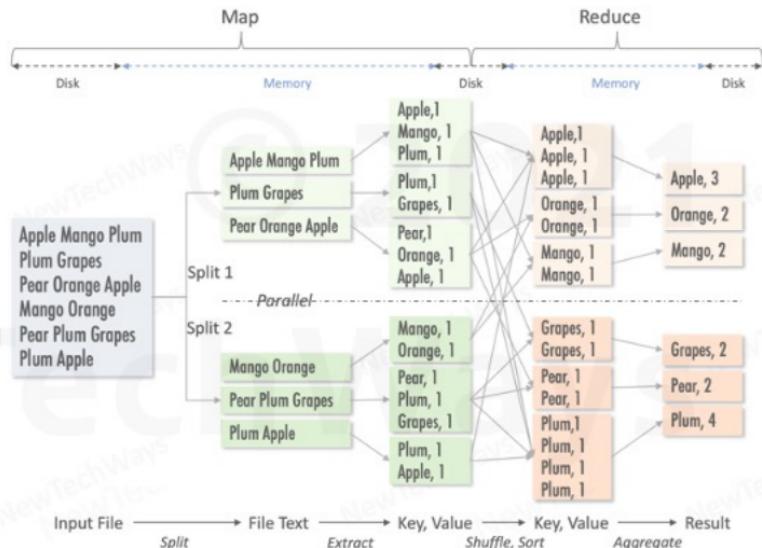


- Client needs to interact with the Master node to find the information of the node and then it can directly talk to the nodes.
- This way master node does not become a bottle neck.

MapReduce:

## Map-Reduce

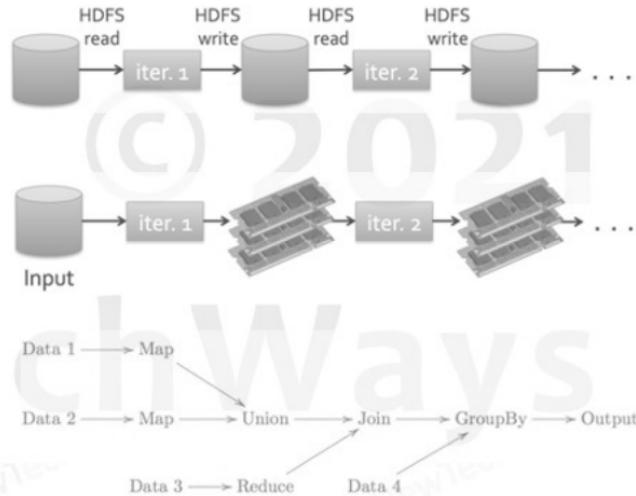
- Parallel file processing on Hadoop cluster
- Processing code executes on datanodes
- Input/Output sources
  - HDFS, HBase, Cassandra
- Map phase
  - Filtering and transformation
- Reduce phase
  - Shuffles map output across nodes
  - Groups related information
  - Computes aggregate data
- Input source to MapReduce can be from HDFS or HBase or Cassandra but the intermediate step for storing into disk should be Hadoop Cluster.
- Used for bulk processing which can take hours or days or significant amount of time.



Apache Spark:

## Apache Spark

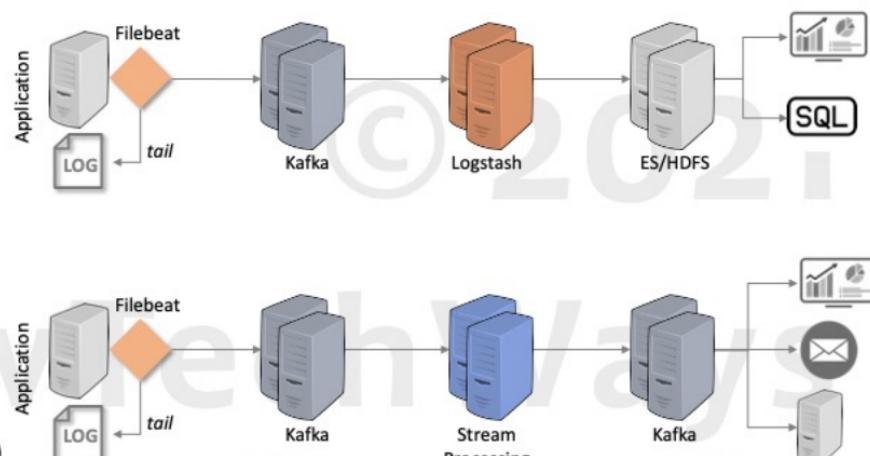
- Evolution of Map-Reduce
- In memory
  - 10x to 100x times faster
- DAG of operations
  - Multiple operations in a DAG
  - Multiple inbuilt operations
- Interactive
  - Interactive shell in Scala/Python/R
- In built libraries for
  - SQL Interface, Machine Learning, Graph Processing, Streaming
- In case of Hadoop MapReduce data is read from disk to memory and then after map function it is again written onto disk and then again when reduce function starts data is read into memory and then again written into disk.
- In case of Apache Spark the data is kept in memory during the intermediate steps rather than writing into disk which speeds up the process by 10 – 100x.



Stream Processing:

## Streaming Processing

- Low latency
  - Keep data moving
- High throughput
  - Multiple sources
- Event Issues
  - Event delay
  - Out-of-Order
  - Missing
- Fault Tolerance
  - Event Replay
- High Availability
- Processing Engine
  - Storm
  - Flink
  - Spark (Micro-Batching)
- Buffer
  - Kafka

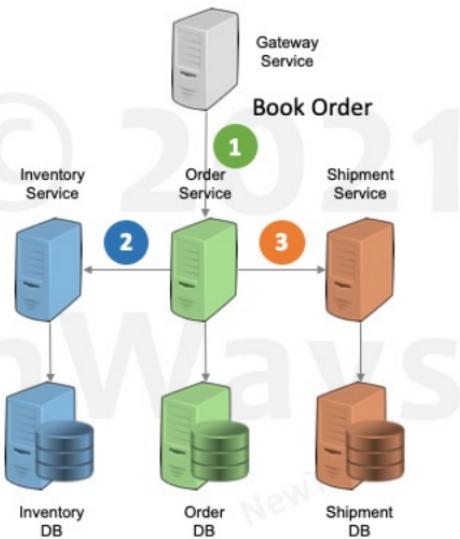


When we want analysis of data in real time like fraud detection, we do stream processing.

## Transactions in Microservices

# Micro-Services Transactions

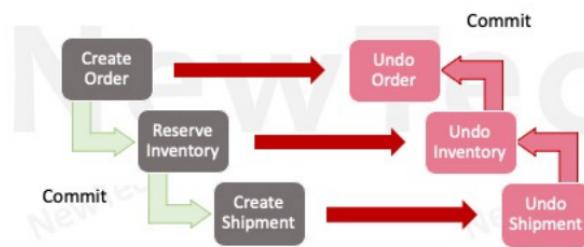
- Transaction involves multiple machines
  - Distributed services with their own DB
  - Local transaction not possible
- Options
  - Distributed ACID Transactions
    - 2PC/3PC
    - Completely ACID
  - Compensating Transactions
    - SAGA pattern
    - Eventually consistent model
      - Relaxes consistency and isolation



SAGA Pattern:

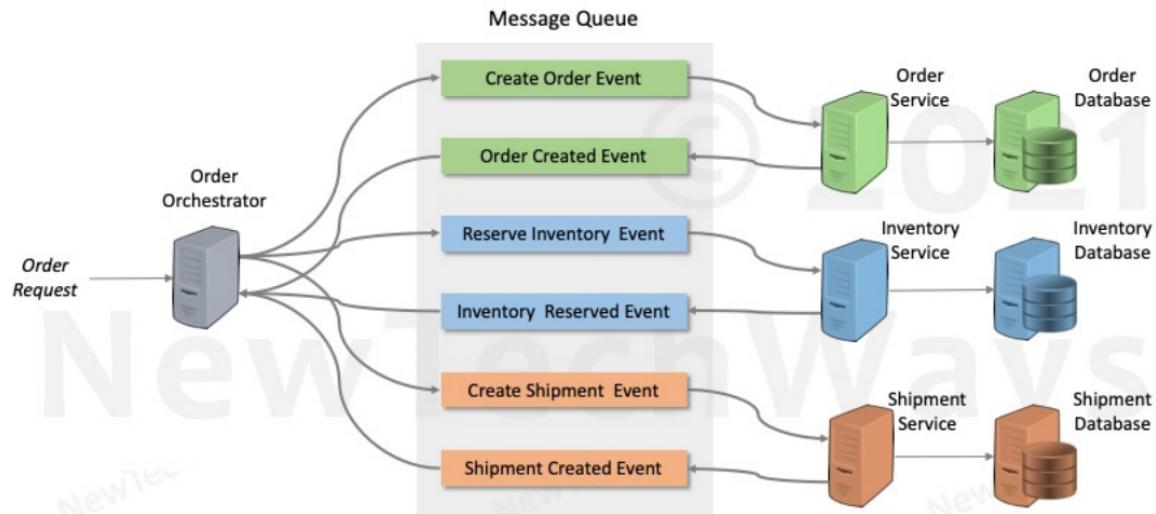
## Compensating Transactions – SAGA Pattern

- ‘Logical Undo’ of a partially committed transactions
  - Flow of reversal may not be exactly opposite, and some steps can be executed in parallel
  - Compensation itself can fail. Should be able to restart itself and retry
- Asynchronous processing for reliability



Microservices Event Driven Transactions:

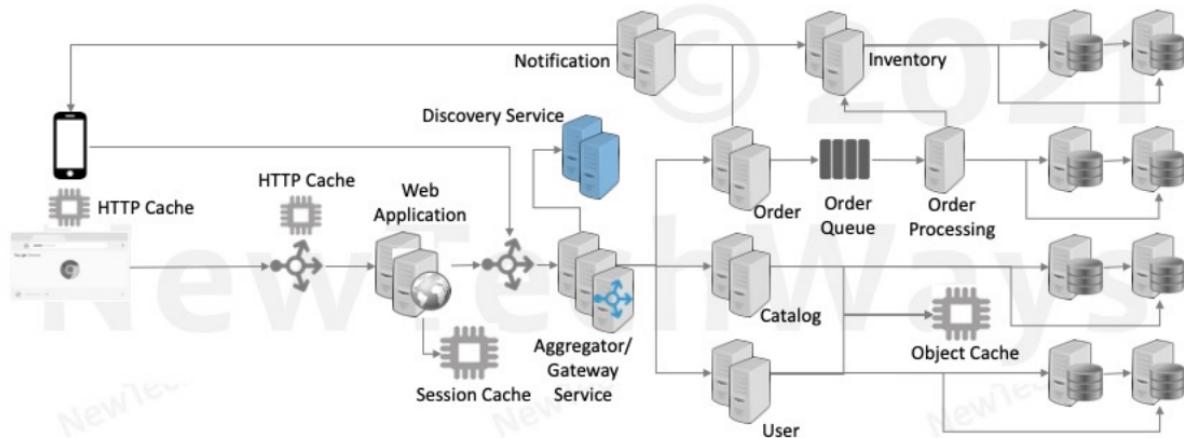
## Micro-Services Event Driven Transactions



Discovery Service & Load Balancing:

## Discovery Service & Load Balancing

- Discovery – Registry for IP of Healthy Instances



- Gateway Service has an embedded router and checks the discovery service for the list of the IPs for a service.
- Based on the algorithm in the router the requests are routed.