High Performance Caching of Travel Inventory Data

Research Documentation

Contributors:
Jarred Wilson
Malcolm Frank
Abebe Adamu
Robert Montgomery
Macauley Odinaka
Dickson Diku

TABLE OF CONTENTS

I. DEFINING THE PROBLEM
II. AVAILABLE TECHNOLOGIES
    A. HAZELCAST
        1. SPECS
            a) SPEED
            b) FAILOVER
            c) LOAD BALANCING
            d) MULTITHREADED
        2. WILL IT WORK?
    B. TIMES TEN
        1. SPECS
            a) SPEED
            b) FAILOVER
            c) LOAD BALANCING
            d) MULTITHREADED
        2. WILL IT WORK?
    C. AEROSPIKE
        1. SPECS
            a) SPEED
            b) FAILOVER
            c) LOAD BALANCING
            d) MULTITHREADED
        2. WILL IT WORK?
    D. GEMFIRE
        1. SPECS
            a) SPEED
            b) FAILOVER
            c) LOAD BALANCING
            d) MULTITHREADED
        2. WILL IT WORK?
    E. CASSANDRA
        1. SPECS
            a) SPEED
            b) FAILOVER
            c) LOAD BALANCING
            d) MULTITHREADED
        2. WILL IT WORK?
III. DESIGN SOLUTION
    A. Platform
    B. Performance

DEFINING THE PROBLEM

In-memory database systems are used to be able to access data more quickly than retrieving it from the disk. The reduction in latency is critical to databases that have high demands on traffic. There are three main requirements to building a production-ready in-memory caching system: speed, failover, and load balancing. Not every caching system has the same approach to meeting these requirements, but they typically support a common feature set that is needed to meet the industry's needs.

SPEED

Overall speed requirements of the in-memory database caching system is dictated by the projected load of the system. Their current system processes an estimated 10 billion requests per month. It is predicted that the workload in the near future will be roughly 100 billion requests per month (38,580 requests per second). Modern database systems use a clustered approach to support the scalability needed to meet the speed requirements.

FAILOVER

A very important component of any system in production is the ability to be able to recover from a system failure. There are several ways to implement failover. In clustered systems, it is common to have redundant nodes in the cluster that can take over for a failed node. This method is seen in the sponsors current caching system.

LOAD BALANCING

A secondary function of a cluster is to provide a level of load balancing on the systems. Some caching systems implement advanced load balancing techniques while others keep it primitive. When specific data in the cache is used much more frequently than others and no advanced load balancing technique is implemented, what is called a hotspot is created. These hotspots are nodes that have too much traffic and not enough throughput.

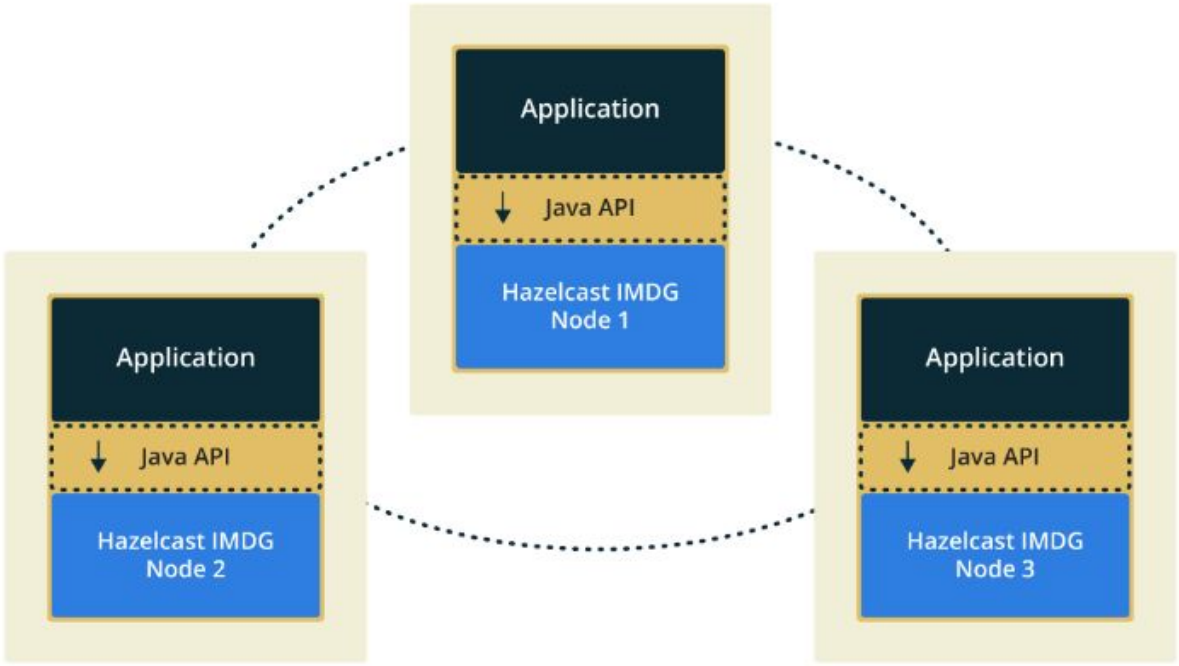
Our sponsor currently uses Redis as their in-memory caching system. Redis meets the speed requirements and supports failover, but it does not take a sophisticated approach to load balancing. The result is hotspots in the cluster. You can technically mitigate the is by identifying the keys that are creating the hotspot and shared them into new nodes to reduce load, but we are going to look at at more sophisticated approaches available in other technologies.

AVAILABLE TECHNOLOGIES

HAZELCAST

Hazelcast is an open source in-memory data grid that provides a **shared nothing** architecture that shares and distributes data across a cluster of servers.
Hazelcast provides multiple ways for you u

HAZELCAST EMBEDDED ARCHITECTURE



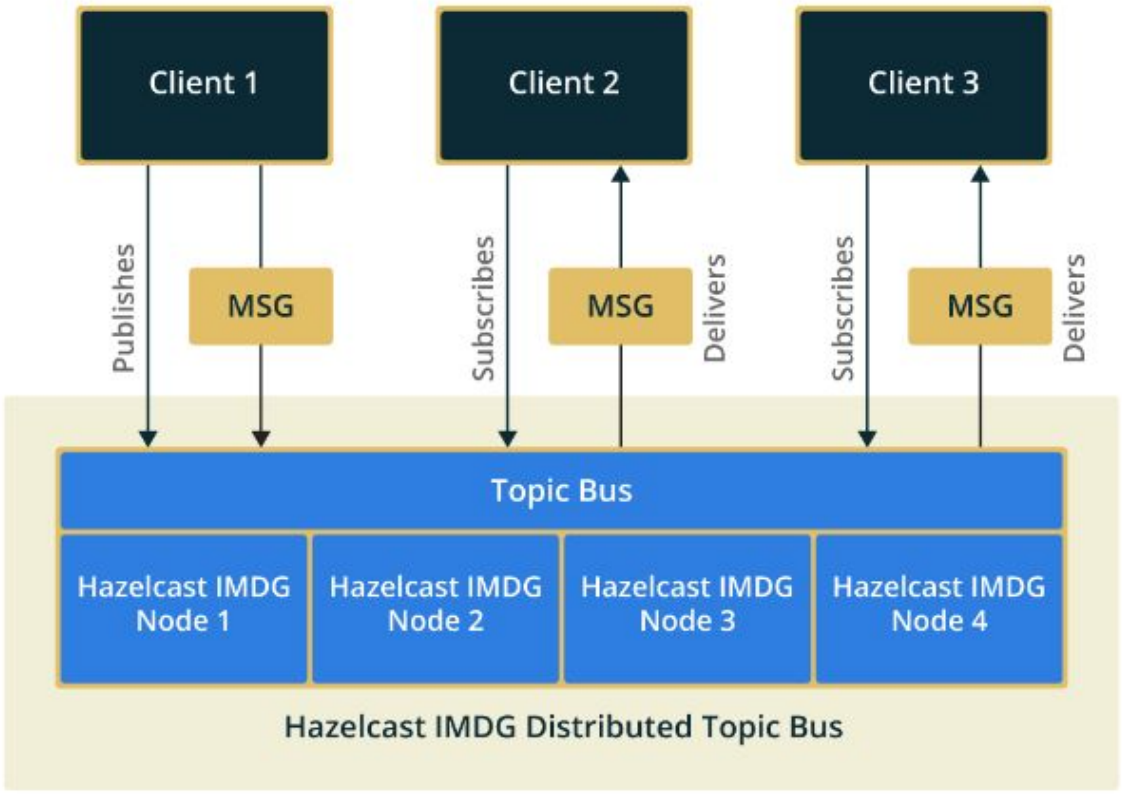
You can use client-server architecture:

- To access Hazelcast from Java, .NET and C++ applications
- To separate the application from Hazelcast for better manageability

se IMDG, depending on your deployment strategies, security aspects, or usage patterns. You can use Hazelcast as a client-server or an embedded architecture.

- **TOPOLOGY**

HAZELCAST CLIENT-SERVER ARCHITECTURE



With both client-server and embedded architecture, Hazelcast offers:

- Elasticity and scalability
- Transparent integration with backend databases using Map Store interfaces
- Management of the cluster through your web browser with Management Center

    - **Shared Nothing Architecture:**
        - Hazelcast clusters do not have a master member. Each cluster member is configured to be the same in terms of functionality. This peer-to-peer network relationship with connections to all members is optimal for speed.

The information provided in this section was derived from the source(s) below:
https://hazelcast.com/use-cases/imdg/

SPECS

SPEED

The images below capture the 2016 benchmark test results that compares the performance of clustered Redis (v.3.0.7) vs clustered Hazelcast (v.3.6.1).

Most up-to-date benchmark comparisons between a Redis 3.2.8 cluster and a Hazelcast IMDG 3.8 cluster can be found at the following link:

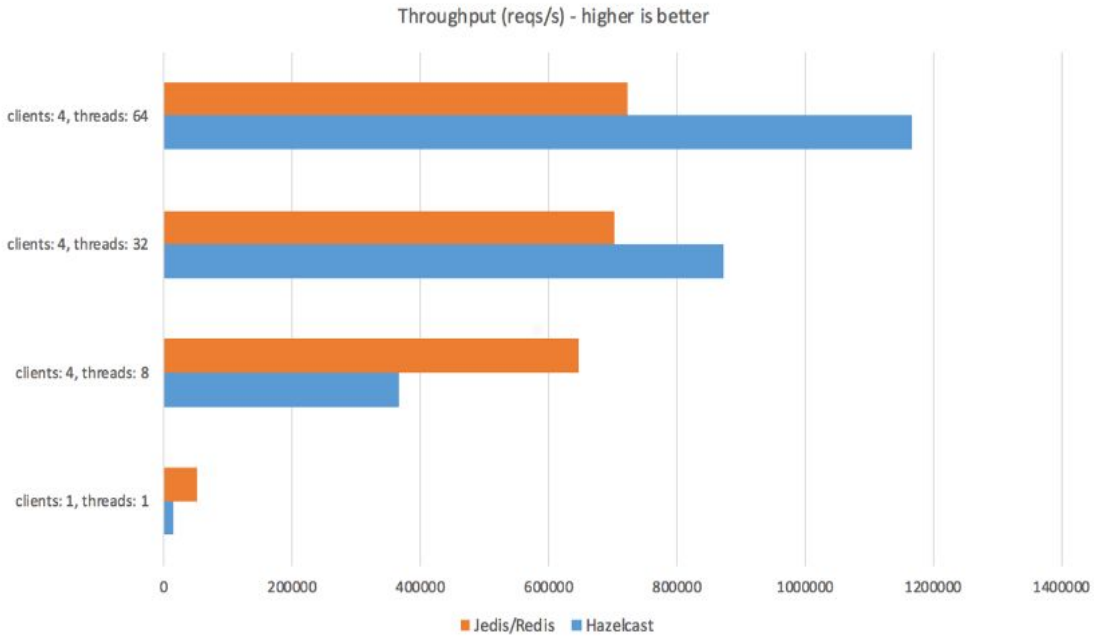
https://hazelcast.com/resources/benchmark-redis-vs-hazelcast/

(Graphs aren't as easy to see and interpret as the ones shown from Redis (v.3.0.7) vs Hazelcast (v.3.6.1)).

- Hazelcast Version 3.6.1
- Redis Version 3.0.7, Jedis 2.8.0

Throughput

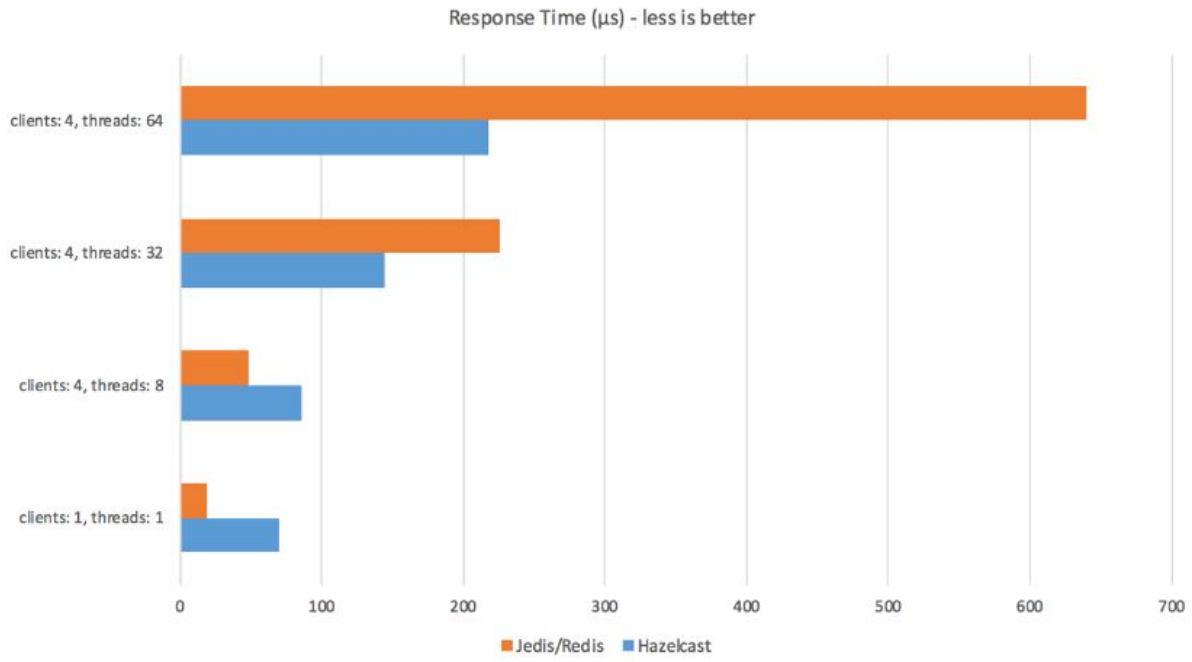
Looking at the throughput results, Redis is extremely fast with a small number of clients and / or threads, however it becomes cumbersome under highly concurrent loads. Scaling beyond a certain number of threads seems to stall the scalability of the internal architecture of Redis. On the other hand, Hazelcast is less impressive under very small loads but scales way beyond when concurrent access and high numbers of clients or threads come into the game.



Scenario	Hazelcast result (get reqs/sec)	Redis result (get reqs/sec)
1	13,954	50,634
2	365,792	645,976
3	872,773	702,671
4	1,166,204	722,531

Latency

According to the results, we seem to have a common Latency performance pattern similar to the Throughput measurements. Redis, under low data load, seems to have a fairly better response time over Hazelcast which reverses when the load and concurrency of the requestors increases. Under load that is not uncommon in big environments (as given in scenario 4) we see Redis dramatically increasing the average response time. Hazelcast response times also grow relative to the number of threads but this increase seems more stable and not exponential as shown by the Redis graph.



Scenario	Hazelcast (resp-time in µs)	Redis (resp-time in µs)
1	70,14	19,37
2	85,83	48,98
3	144,7	225,22
4	217,52	640,51

FAILOVER

Hazelcast shards are called Partitions. By default, Hazelcast has 271 partitions. The partitions themselves are distributed equally among the members of the cluster. Hazelcast also creates the backups of partitions and distributes them among members for redundancy.

Partitions are memory segments that can contain hundreds or thousands of data entries each, depending on the memory capacity of your system. Each Hazelcast partition can have multiple replicas, which are distributed among the cluster members. One of the replicas becomes the primary and other replicas are called backups. Cluster member which owns primary replica of a partition is called partition owner. When you read or write a particular data entry, you transparently talk to the owner of the partition that contains the data entry.

How is data partitioned?

Hazelcast distributes data entries into the partitions using a hashing algorithm. Given a object key or an object name:

we serialize (convert into a byte array), hash the byte array and mod the hash result with the number of partitions

The result of - *mod(hash result, partition count)* - is the partition in which the data will be stored. This partition is considered the partition ID. For all members that are in the cluster, the partition ID for a given key will always be the same.

Partition table

When you start a member, a partition table is created within it. This table stores the partition IDs and the cluster member to which they belong to.

The oldest member (first member created in the cluster) periodically sends the partition table to all cluster members. In this way each member is informed about any changes to partition ownership.

For example, partition ownership would change if the oldest member dies. Subsequently, the second to oldest member would take over.

By default, Hazelcast clients will connect to all members and know about the cluster partition table to route requests to the correct node, preventing additional/unnecessary trips.

The information provided for this section was derived from the source(s) below:
(http://docs.hazelcast.org/docs/latest-development/manual/html/Hazelcast_Overview/Data_Partitioning.html)

LOAD BALANCING

There are two situations where the balance between the number of threads performing operations and the number of operations being executed can become disproportionate:

Asynchronous calls: With async calls, the system may be flooded with new requests.

Asynchronous backups: The asynchronous backups may be piling up.

To prevent the system from crashing, Hazelcast provides back pressure. Back pressure works by:

- Limiting the number of concurrent operation invocations
- Periodically making an async backup sync

Memberside:

Back pressure is disabled by default, to enable it we need to use the following system property:

hazelcast.backpressure.enabled

To control the number of concurrent invocations, you can configure the number of invocations allowed per partition using the following system property:

hazelcast.backpressure.max.concurrent.invocations.per.partition

Client Side:

To prevent the system at the client side from overloading, we can apply a constraint on the number of concurrent invocations. We can use the following system property at the client side for this purpose:

Hazelcast.client.max.concurrent.invocations

The information provided for this section was derived from the source(s) below:
(http://docs.hazelcast.org/docs/latest-development/manual/html/Performance/Back_Pressure.html)

MULTITHREADED

Hazelcast uses highly optimized, multithreaded clients and servers, and uses asynchronous I/O, meaning that each thread owns its partitions, so there is no contention.

On each cluster member, the I/O threading is split up in 3 types of I/O threads:

- I/O thread for the accept requests
- I/O thread to read data from the other members/clients

- I/O thread to write data to other members/clients

Hazelcast periodically scans utilization of each I/O thread and can decide to migrate a connection to a new thread if the existing thread is servicing a disproportionate number of I/O events.

The scanning interval is customizable through configuring the *hazelcast.io.balancer.interval.seconds* system property. (default interval is 20 seconds)

The information provided in this section was derived from the source(s) below:
(http://docs.hazelcast.org/docs/latest-development/manual/html/Performance/Threading_Model/I:O_Threading.html)
(https://hazelcast.com/resources/benchmark-redis-vs-hazelcast/)

WILL IT WORK?

Hazelcast is Simple

Hazelcast is written in Java with no other dependencies. It exposes the same API from the familiar Java util package, exposing the same interfaces. Just add hazelcast.jar to your classpath and you can quickly enjoy JVMs clustering and start building scalable applications.

Hazelcast is Peer-to-Peer

Unlike many NoSQL solutions, Hazelcast is peer-to-peer. There is no master and slave; there is no single point of failure. All members store equal amounts of data and do equal amounts of processing. You can embed Hazelcast in your existing application or use it in client and server mode where your application is a client to Hazelcast members.

Hazelcast is Scalable

Hazelcast is designed to scale up to hundreds and thousands of members. Simply add new members and they will automatically discover the cluster and will linearly increase both memory and processing capacity. The members maintain a TCP connection between each other and all communication is performed through this layer.

Hazelcast is Fast

Hazelcast stores everything in-memory. It is designed to perform very fast reads and updates.

Hazelcast is Redundant

Hazelcast keeps the backup of each data entry on multiple members. On a member failure, the data is restored from the backup and the cluster will continue to operate without downtime.

Hazelcast's Distinctive Strengths:

Hazelcast is open source.

Hazelcast is only a JAR file. You do not need to install software.

Hazelcast is a library, it does not impose an architecture on Hazelcast users.

Hazelcast provides out-of-the-box distributed data structures, such as Map, Queue, MultiMap, Topic, Lock and Executor.

There is no "master," meaning no single point of failure in a Hazelcast cluster; each member in the cluster is configured to be functionally the same.

When the size of your memory and compute requirements increase, new members can be dynamically joined to the Hazelcast cluster to scale elastically.

Data is resilient to member failure. Data backups are distributed across the cluster. This is a big benefit when a member in the cluster crashes as data will not be lost.

Members are always aware of each other unlike in traditional key-value caching solutions.

You can build your own custom-distributed data structures using the Service Programming Interface (SPI) if you are not happy with the data structures provided.

The information provided for this section pertains to the source(s) below:
(http://docs.hazelcast.org/docs/latest/manual/html-single/)

(http://docs.hazelcast.org/docs/latest-development/manual/html/Hazelcast_Overview/Why_Hazelcast.html)

TIMES TEN

Oracle Time Ten is a in-Memory Database technology full-featured , memory-optimized, relational database with persistence and recoverability. Times Ten provides instant responsiveness and very high throughput required by database-intensive applications. Times Ten operates on databases that fit entirely in physical memory(RAM). Times Ten is deployed as an in-memory cache database with automatic data synchronization between Times Ten and the Oracle Database.

SPECS

SPEED

Times Ten provides high performance and data replication with no exception. The figure below shows a workload with a single stream replication that can sud=stain a maximum throughput of around 60,000 transactions per second. Enabling parallel replication (with parallelism = 8) delivers a peak throughput of almost 180,000 TPS (almost 3x that of single stream replication) using the default setting whereby commit ordering is enforced. Enabling the 'no commit ordering' optimization shows even better results with the peak throughput increasing to 285,000 TPS (nearly 5x that of single stream replication). Data are stored in memory, and accessed at RAM speeds instead of network speeds, TimesTen can provide extremely fast response time...

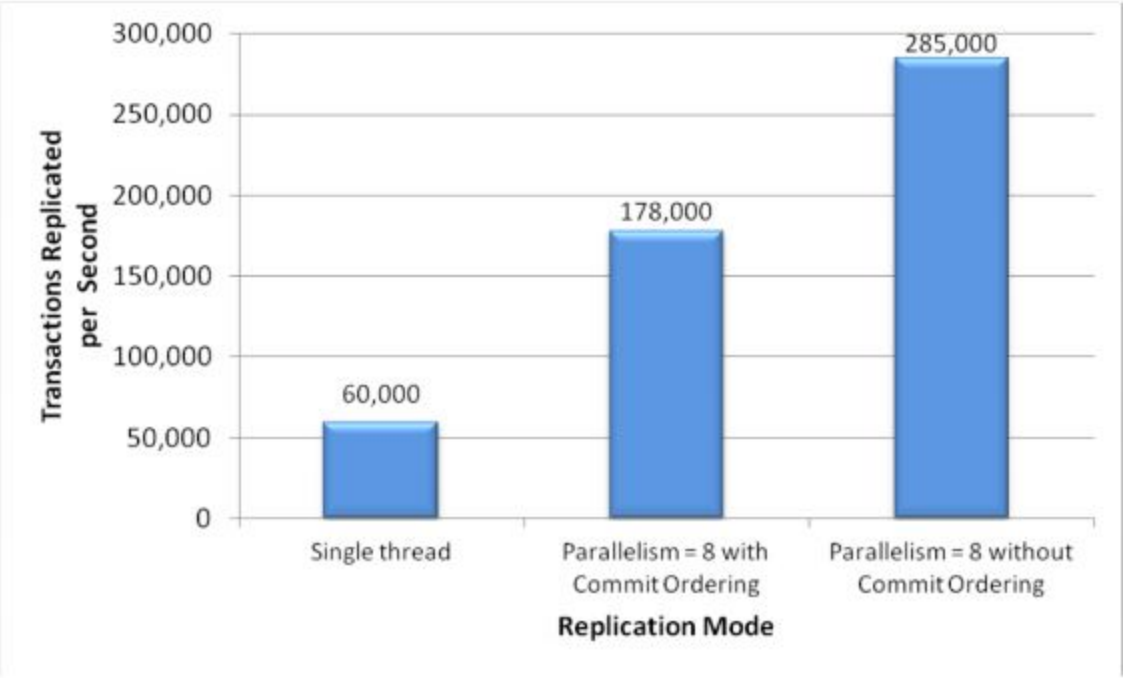


Figure 5: Asynchronous replication throughput for committed singleton updates

TimesTen can achieve response times in the microseconds due to its in-memory architecture. With TimesTen, a transaction that reads a database record can take fewer than 2.5 microseconds, and transactions that update or insert a record can take fewer than 8 microseconds.

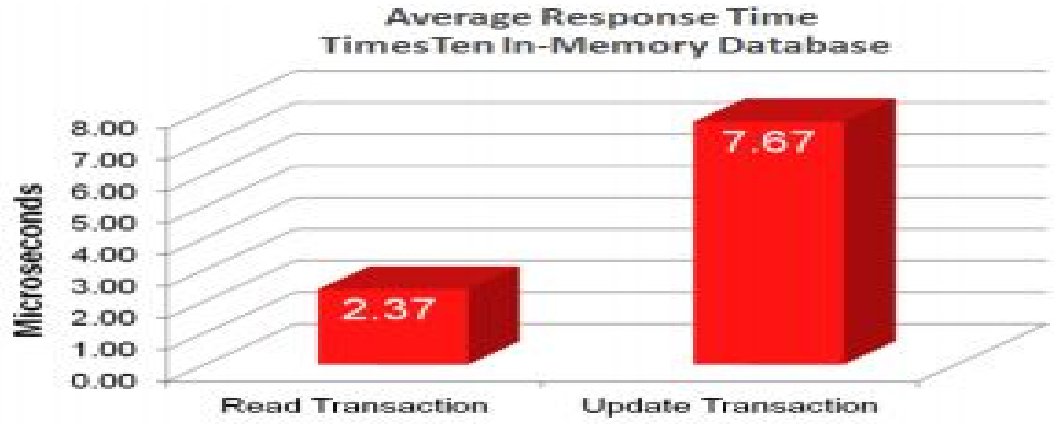


Figure 2. TimesTen Response Times

Figure 2 shows the response times for an application executing read and update transactions on an Intel E5-2680 @2.7GHz 2 sockets 8cores/socket system running Oracle Linux.
The graph below shows a significant reduction in application response time when using TimesTen Cache

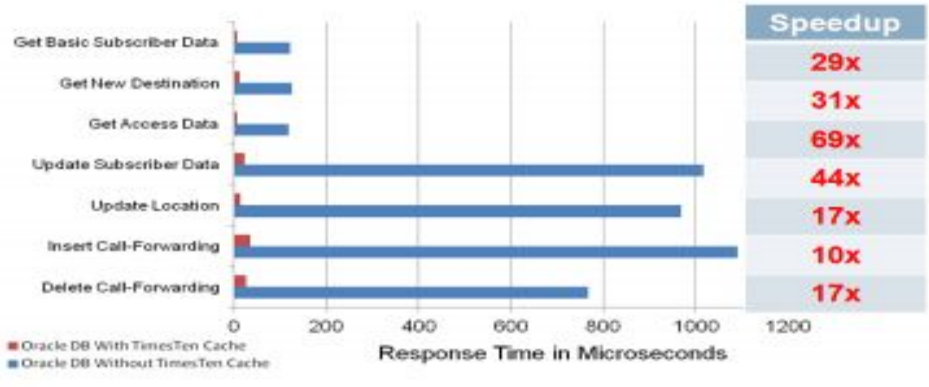


Figure 10. Response Time Comparison for HLR Benchmark Application

Time Ten Scalability

TimesTen takes advantage of multiple CPUs on symmetric multiprocessor computers. Figure 3 shows the transaction throughput of TimesTen on an Intel E5-2680 @2.7GHz 2 sockets 8 cores/socket system running Oracle Linux. Each transaction executes a single SQL select (read) or update operation, as indicated. The results are shown for 1, 4, 8 and 16 concurrent processes.

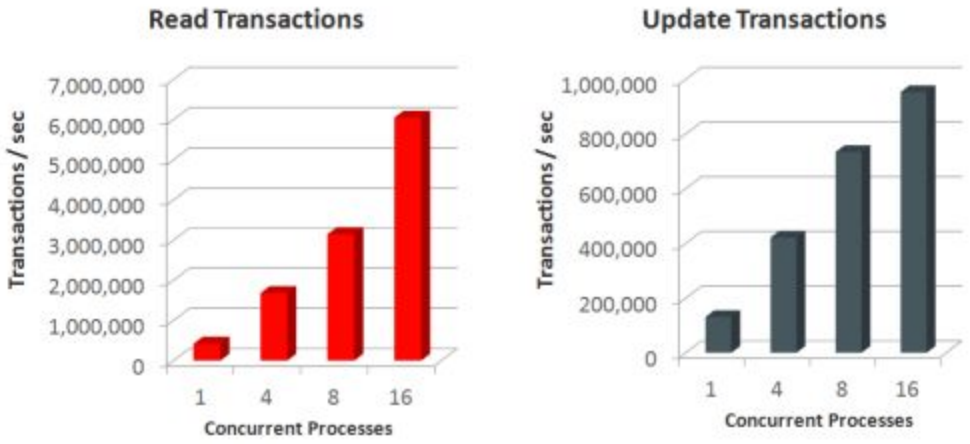


Figure 3. The throughput of TimesTen

Time Ten highest volume workload is 100% reads tes, and measures many individual records that can be retrieved using a key lookup. Update operations require more processing than read operations, partly because changes must be logged for recovery purposes. In this result, more than 952,778 records were updated per second with 16 concurrent processes, and more than 130,375 with a single process.

FAILOVER

TimesTen Cache will failover automatically to the standby Oracle database without data loss.
If the Oracle Database becomes inaccessible to TimesTen Cache for any reason such as network failure, hardware failure, or Oracle Database failure, TimesTen Cache is designed to be resilient to such failures. The in-memory cache will continue to be accessible to applications. Furthermore, in case of an AWT Cache Group, updates to the cache will continue to be logged in Oracle TimesTen so that once the Oracle Database becomes accessible again, the updates are propagated to it. Similarly changes to Read-Only Cache Groups that were made on the Oracle Database but not yet propagated to the in-memory cache will remain recorded on the Oracle database and will be propagated to the cache(s) once the Oracle database is accessible again.

An active standby pair includes an active master database, a standby master database, and optional read-only subscriber databases. The active master database is updated directly. The standby master database cannot be updated directly. It receives the updates from the active master database and propagates the changes to read-only subscriber databases. This arrangement ensures that the standby master database is always ahead of the read-only subscriber databases and enables rapid failover to the standby database if the active master database fails. Only one of the master databases can function as an active master database at a specific time. If the active master database fails, the role of the standby master database must be changed to active before recovering the failed database as a standby database. The replication agent must be started on the new standby master database. If the standby master database fails, the active master database replicates changes directly to the read-only subscribers. After the standby master database has recovered, it contacts the active standby database

to receive any updates that have been sent to the read-only subscribers while the standby was down or was recovering. When the active and the standby master databases have been synchronized, then the standby resumes propagating changes to the subscribers. Automatic failure detection and failover of database and applications is available through integration with Oracle Clusterware. Active standby replication can be used with the Oracle TimesTen Application-Tier Database Cache to achieve cross-tier high availability. Active standby replication is available for both read-only and updatable caches.

LOAD BALANCING

TimesTen Cache takes full advantage of RAC's high availability features. A RAC configuration consists of a single physical database that is accessible by several nodes. The runtime configuration on a single node is called an instance. RAC provides load balancing, high availability, and data consistency across all instances.
The replication mechanism in TimesTen is by default asynchronous. When using asynchronous replication, an application updates a master database and continues working without waiting for the updates to be received by the subscribers. The master and subscriber databases have internal mechanisms to confirm that the updates have been successfully received and committed by the subscriber. These mechanisms, which are completely invisible to the application, ensure that updates are not lost and are applied by a subscriber only once.

MULTITHREADED

TimesTen transactions conform to the atomicity, consistency, isolation, and durability properties. These properties ensure that, in a multiuser system, each transaction operates as if it were the only transaction being executed at the time, and that the system can guarantee that the effects of a committed transaction are not lost. These are the most rigid properties required of data managers, and TimesTen ensures full conformance. A common misperception is that in-memory data managers cannot prevent data loss from system failures. In fact, the same techniques that make transactions and data durable in a conventional database are used in TimesTen. As in all transaction-oriented systems, durability is achieved through a combination of change logging and periodic refreshes of a version of the database that resides on a disk.

WILL IT WORK?

Times Ten provides application-tier data management for performance-critical systems, optimized for blazing-fast response and real-time caching of Oracle data. Thousands of companies worldwide, including Alcatel-Lucent, Aspect, Avaya, Bank of America Merrill Lynch, Bridgewater Systems, BroadSoft, Cisco, Deutsche Börse, Ericsson, JPMorgan, KDDI, NEC, NYFIX, Smart Communications, United States Postal Office and Verizon Wireless use Oracle TimesTen in production applications.

AEROSPIKE

SPECS

SPEED

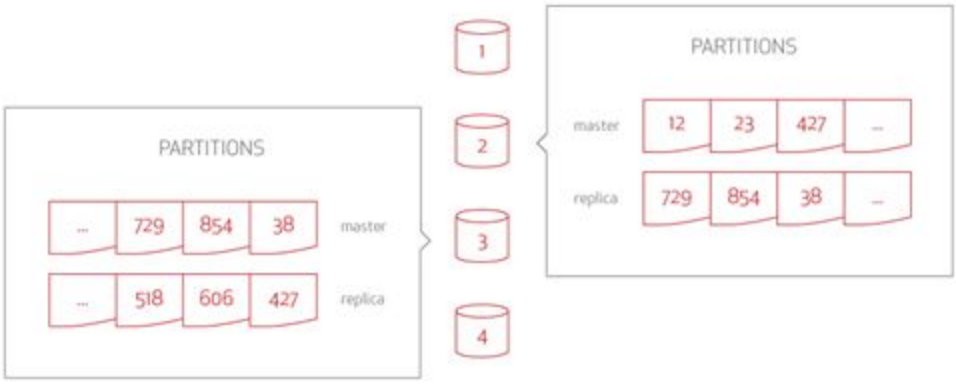
Aerospike scales linearly on the horizontal axis and each node is capable of processing up to 1M TPSz the aggregate cluster top speed is built for very big and fast data. Most companies need only a fraction of this speed, but this top speed has benefit of increased stability and predictability at lower speeds. Not only can Aerospike handle 1 Million TPS per second on a single node, it can horizontally scale.

FAILOVER

Aerospike reliability guarantee cluster failure detection. On failure of one cluster node, Aerospike immediately recovers and reforms the cluster.Cluster nodes check each other by using the heartbeat feature and heartbeat helps nodes coordinate themselves. There is no masters node and all nodes are peers, and they track each other in the cluster.

LOAD BALANCING

Aerospike in-memory database uses a Shared-Nothing architecture, where every node in the Aerospike cluster is identical meaning that all nodes are peers and there is no single point of failure. Partition are distributed evenly across cluster nodes. For example, if n nodes in the cluster, each node stores 1/n of the data. Data distributes evenly across the cluster nodes which means that there are no hotspots where one node handles a lot more requests than other nodes in the cluster. Aerospike ensures that random data assignment balance server load. For instance, if many last names starts with R, and R has a lot of traffic than the server handling last names beginning with X,Y or Z. Random data assignment ensures a balanced server load. In reliability, Aerospike replicates partitions on one or more nodes. One node becomes the data master for reads and writes for a partition, and others nodes store replicas.



MULTITHREADED

Aerospike is multi-threaded, multi-core, high performance NoSQL solution. One can run several high throughput operations in parallel with 99% response times that are less than one millisecond.

WILL IT WORK?

Yes, I think Aerospike will work because all nodes are peers, and data are distributed randomly and evenly across the cluster nodes.

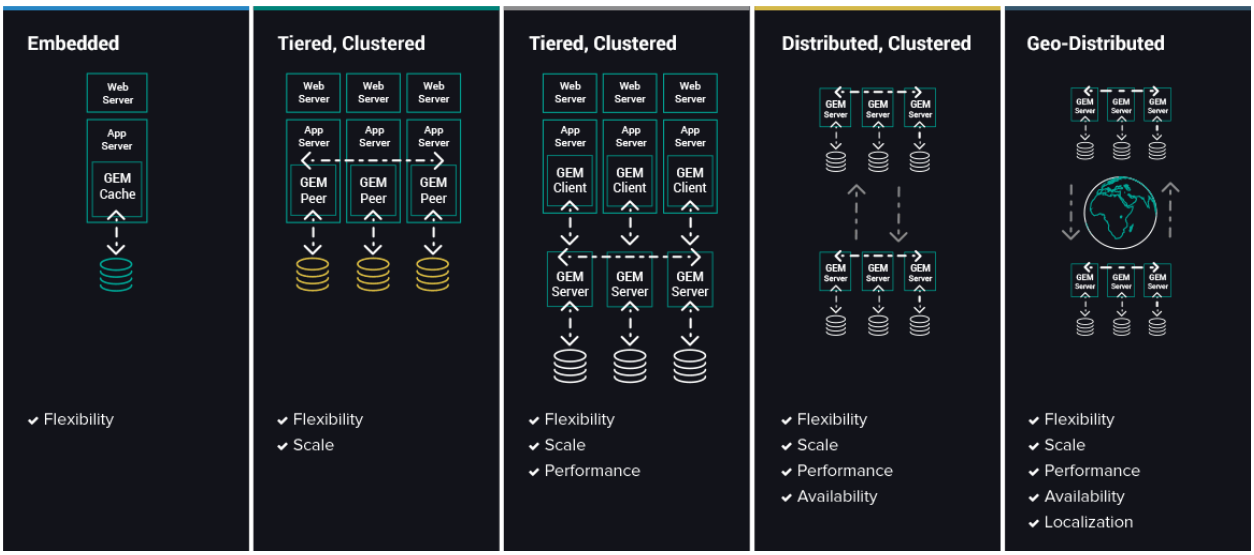
GEMFIRE

Gemfire is a multithreaded in-memory data grid that is powered by Apache. This data grid has multiple features including but is not limited to:

- Predictable low latency
- Elastic Scale-Out
- Real-Time Event Notifications
- High Availability and Business Continuity
- Durability
- Cloud-Ready

Gemfire's low latency feature allows it to perform at high speed levels.

Gemfire is very flexible and can be deployed in many different kinds of configurations.



SCALABILITY

Scalability is achieved through dynamic partitioning of data across many members and spreading the data load uniformly across the servers. For "hot" data, you can configure the system to expand dynamically to create more copies of the data. You can also provision application behavior to run in a distributed manner in close proximity to the data it needs. If you need to support high and unpredictable bursts of concurrent client load, you can increase the number of servers managing the data and distribute the data and behavior across them to provide uniform and predictable response

times. Clients are continuously load balanced to the server farm based on continuous feedback from the servers on their load conditions. With data partitioned and replicated across servers, clients can dynamically move to different servers to uniformly load the servers and deliver the best response times. You can also improve scalability by implementing asynchronous “write behind” of data changes to external data stores, like a database. This avoids a bottleneck by queuing all updates in order and redundantly. You can also conflate updates and propagate them in batch to the database.

FAILOVER

In a multi-site installation, if the primary gateway server goes offline, a secondary gateway sender must take over primary responsibilities as the failover system. The existing secondary gateway sender detects that the primary gateway sender has gone offline, and a secondary one becomes the new primary. Because the queue is distributed, its contents are available to all gateway senders. So, when a secondary gateway sender becomes primary, it is able to start processing the queue where the previous primary left off with no loss of data.

LOAD BALANCING

The locator is a GemFire process that tells new, connecting members where running members are located and provides load balancing for server use. You can run locators as peer locators, server locators, or both: Peer locators give joining members connection information to members already running in the locator’s distributed system. Server locators give clients connection information to servers running in the locator’s distributed system. Server locators also monitor server load and send clients to the least-loaded servers. By default, locators run as peer and server locators. You can run the locator standalone or embedded within another GemFire process. Running your locators standalone provides the highest reliability and availability of the locator service as a whole.

WILL IT WORK?

Yes, Gemfire will work because it distributes and replicates data to many nodes, while maintaining high-performance speeds.

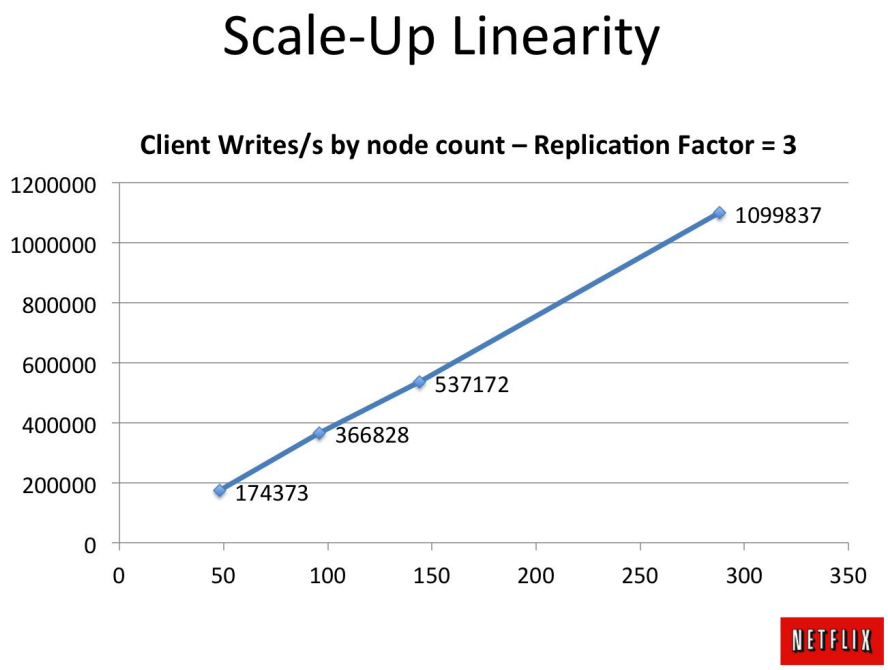
CASSANDRA

Apache Cassandra is a key-value store and widely used in large-scale real-time internet applications, such as Netflix, Reddit, The Weather Channel.

Speed

Netflix had a test run designed to validate their tooling and automation scalability as well as the performance characteristics of Cassandra.The scalability was linear as shown in the chart below. Each client system generates about 17,500 write requests per second, without bottlenecks. Each client ran 200 threads to generate traffic across the cluster.Benchmarking Cassandra Scalability on AWS resulted Over a million writes per second.The automated tooling that Netflix has a Cassandra cluster consisting of

288 medium sized instances, with 96 instances in each of three EC2 availability zones in the US-East region. Using an additional 60 instances as clients they ran a workload of 1.1 million client writes per second. Data was automatically replicated across all three zones making a total of 3.3 million writes per second across the cluster.



Per-Instance Activity

The average activity level on each instance for each of these tests is shown below.

Per Node Activity

Per Node	48 Nodes	96 Nodes	144 Nodes	288 Nodes
Per Server Writes/s	10,900 w/s	11,460 w/s	11,900 w/s	11,456 w/s
Mean Server Latency	0.0117 ms	0.0134 ms	0.0148 ms	0.0139 ms
Mean CPU %Busy	74.4 %	75.4 %	72.5 %	81.5 %
Disk Read	5,600 KB/s	4,590 KB/s	4,060 KB/s	4,280 KB/s
Disk Write	12,800 KB/s	11,590 KB/s	10,380 KB/s	10,080 KB/s
Network Read	22,460 KB/s	23,610 KB/s	21,390 KB/s	23,640 KB/s
Network Write	18,600 KB/s	19,600 KB/s	17,810 KB/s	19,770 KB/s

Node specification – Xen Virtual Images, AWS US East, three zones

- Cassandra 0.8.6, CentOS, SunJDK6
- AWS EC2 m1 Extra Large – Standard price $ 0.68/Hour
- 15 GB RAM, 4 Cores, 1Gbit network
- 4 internal disks (total 1.6TB, striped together, md, XFS)

NETFLIX

https://medium.com/netflix-techblog/benchmarking-cassandra-scalability-on-aws-over-a-million-writes-per-second-39f45f066c9e

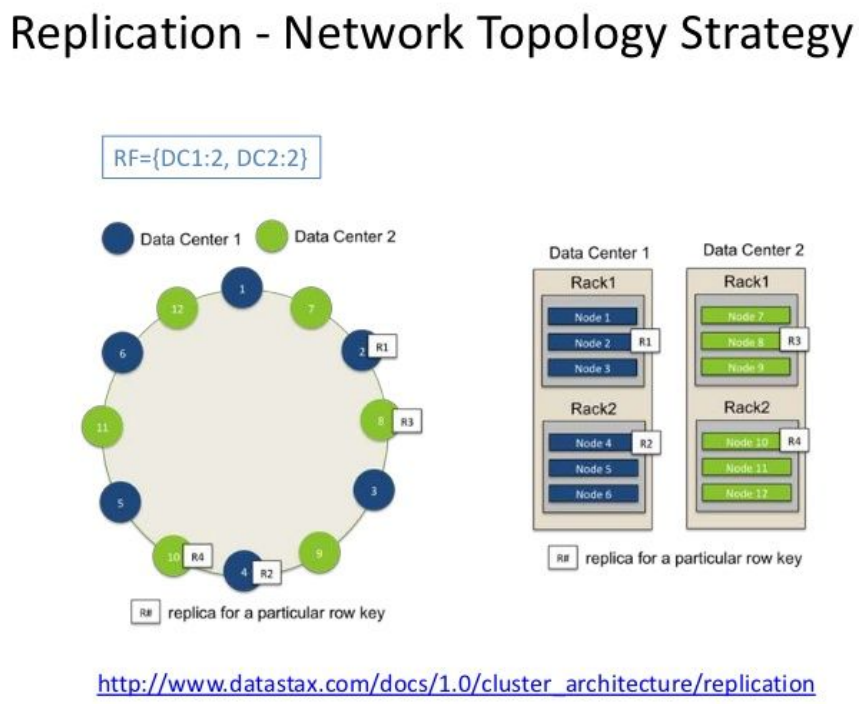
FAILOVER

Cassandra stores replicas on multiple nodes to ensure reliability and fault tolerance. A replication strategy determines the nodes where replicas are placed. The total number of replicas across the cluster is referred to as the replication factor. A replication factor of 1 means that there is only one copy of each row in the cluster. If the node containing the row goes down, the row cannot be retrieved. A replication factor of 2 means two copies of each row, where each copy is on a different node. All replicas are equally important; there is no primary or master replica. As a general rule, the replication factor should not exceed the number of nodes in the cluster. However, you can increase the replication factor and then add the desired number of nodes later.

Two replication strategies are available:

SimpleStrategy: Use only for a single datacenter and one rack. If you ever intend more than one datacenter, use the NetworkTopologyStrategy.

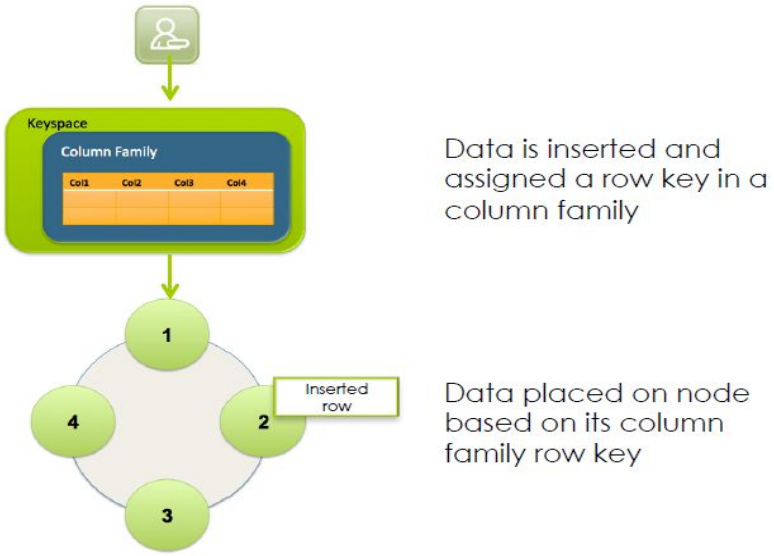
NetworkTopologyStrategy: Highly recommended for most deployments because it is much easier to expand to multiple datacenters when required by future expansion.
https://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archDataDistributeReplication.html



http://www.datastax.com/docs/1.0/cluster_architecture/replication

LOAD BALANCING

Cassandra partitions data across all participating nodes in a database cluster. Each node is responsible for part of the overall database. When data is inserted ,it assigns a row key and based on this particular key cassandra assigns this row to one of the nodes in the cluster which is responsible to manage it.



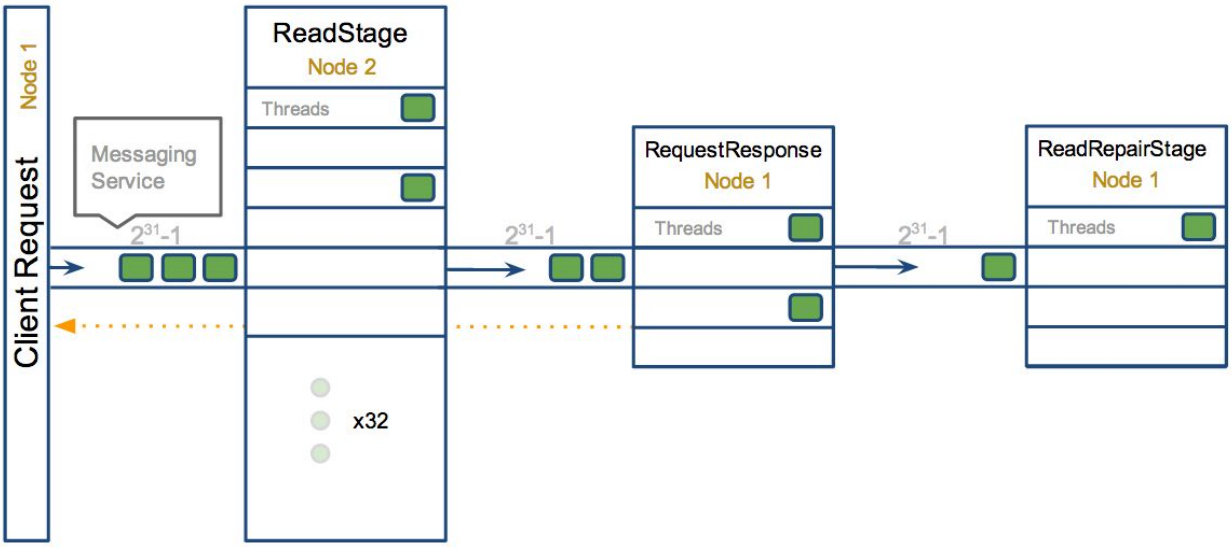
There are two basic partitioning strategies:

1. Random partitioning: this is the default and recommended strategy. partitioning data as evenly as possible across all nodes using an MD5 hash of every column family row key. This one ensures even distribution of data throughout all the nodes in the cluster.
2. Ordered partitioning: stores column family row keys in sorted order across the nodes in a database cluster. This is not a recommended strategy, because a number of different issues can arise such as having **hotspots** in cluster , and not evenly distributed load balance.

https://www.datastax.com/resources/tutorials/partitioning-and-replication

MULTITHREADED

Cassandra is designed for high concurrency so it is multithreaded.Cassandra is based off of a Staged Event Driven Architecture (SEDA). This separates different tasks in stages that are connected by a messaging service. Each like task is grouped into a stage having a queue and thread pool (ScheduledThreadPoolExecutor more specifically for the Java folks). Some stages skip the messaging service and queue tasks immediately on a different stage if it exists on the same node. Each of these queues can be backed up if execution at a stage is being over run. This is a common indication of an issue or performance bottleneck.To demonstrate take for example a read request:



https://blog.pythian.com/guide-to-cassandra-thread-pools/

WILL IT WORK?

Yes , it will work.

Cassandra is **Fault tolerant:** Data is automatically replicated to multiple nodes for fault-tolerance. Replication across multiple data centers is supported. Failed nodes can be replaced with no downtime.

Scalable : Some of the largest production deployments include Apple's, with over 75,000 nodes storing over 10 PB of data, Netflix (2,500 nodes, 420 TB, over 1 trillion requests per day), Chinese search engine Easou (270 nodes, 300 TB, over 800 million requests per day), and eBay (over 100 nodes, 250 TB).

Decentralized: There are no single points of failure. There are no network bottlenecks. Every node in the cluster is identical.

DESIGN SOLUTION
