Partitioning (Sharding) and Replication

Writeup Submissions (/student/submissions/14/91) Scoreboard (/student/scoreboard/14/91) Costs (/student/costs/14/91)

2 days 1 hour left

Show Submission Password

Module	Open	Deadline
Partitioning (Sharding) and Replication	02/29/2016 00:01 -0500	03/06/2016 23:59 -0500

Introduction

Learning Objectives

This project will encompass the following learning objectives:

1. Describe the motivation and design space for distributed key-value stores.
2. Compare and contrast the advantages and disadvantages of using replication and sharding in distributed key-value stores.
3. Extend a distributed key-value store with sharding or replication schemes.
4. Understand consistent hashing and implement a consistent hashing algorithm to illustrate its applicability to sharding in distributed key-value stores.
5. Apply replication and sharding techniques to real-world scenarios.

General Details

The following table contains the general information about this project phase:

Applicable Languages			
• Java only			
Tasks	**Total Budget**	**Bonuses?**	**Cloud Platform**
2	$10	Yes, for a good hash function (read below).	AWS Only

Grading Penalties

Besides the penalties mentioned in recitation and/or on Piazza, penalties accrue for the following:

Violation	Penalty of the project grade
Spending more than $10 for this project checkpoint	-10%

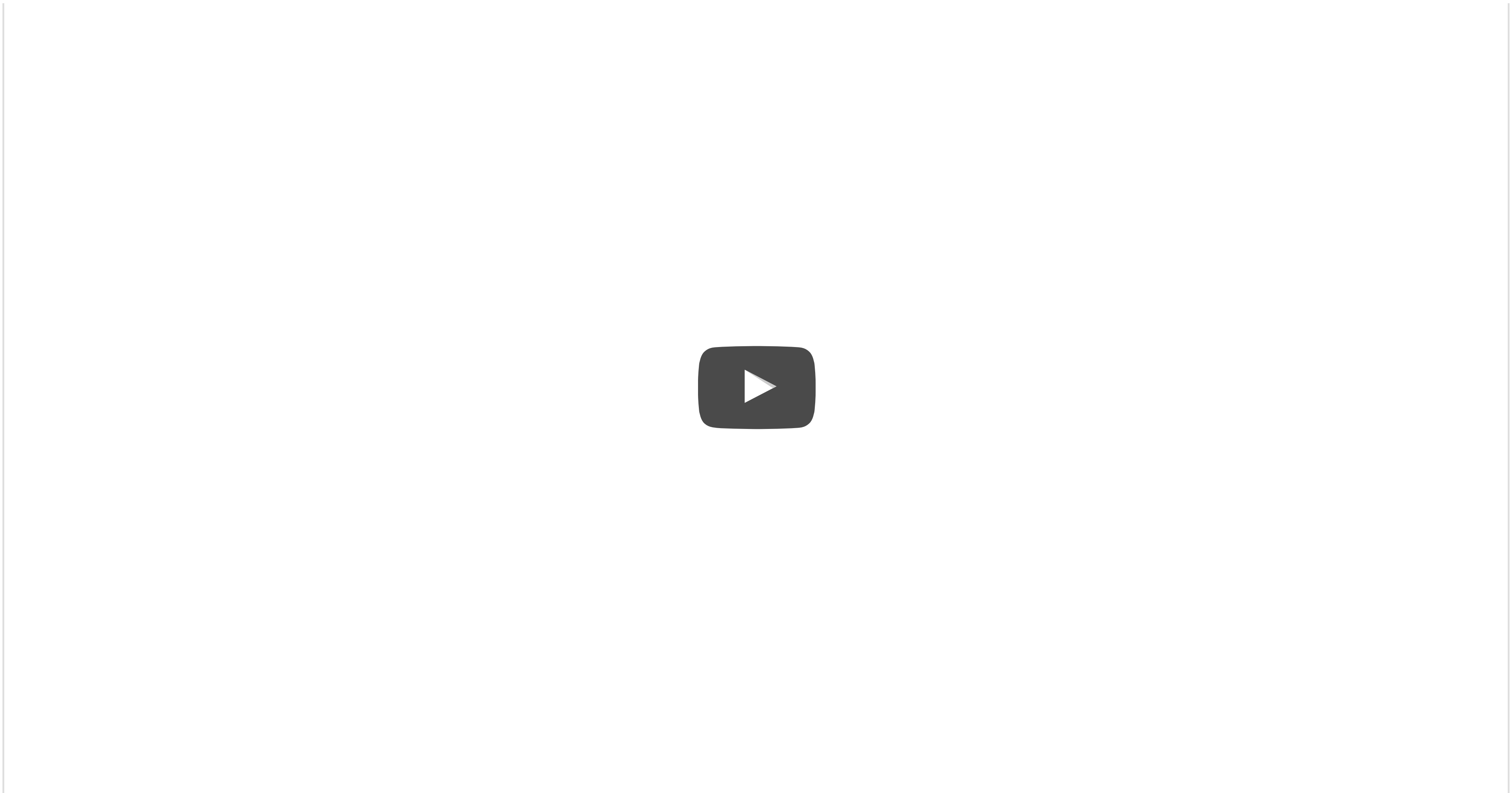
Spending more than $20 for this project checkpoint	-100%
Failing to tag all your resources for this project	-10%
Using any instance other than t1.micro or m1.small	-10%
Attempting to hack/tamper the auto-grader	-100%
Not submitting Coordinator.java for final submission	-100%
Using `String.hashCode()`	-5%

With the advent of the Internet, e-commerce and social media, organizations are facing a massive explosion in the amount of data that needs to be handled daily. It is not uncommon these days for large Internet-scale companies to have to process petabytes of data on a daily basis. Storing, processing and analyzing this data is an enormous challenge, and has long surpassed the storage, memory and computing capabilities of a single machine. We require distributed, scalable data storage systems to handle this big-data challenge. While there are many types of distributed databases and storage systems, in this project we will focus on *distributed key-value stores*, a commonly adopted solution for scaling up data storage. These key-value stores are considered to be a type of *NoSQL* storage system, as they do not have full relational capabilities of systems such as MySQL, which you explored in the previous project. Key-value stores are also commonly used for caching systems for storing recent results for a given key. Unlike SQL, a key-value store supports two basic operations:

1. `PUT` requests, which puts a value for a given key in the database
2. `GET` requests, which gets the value associated with a key in the database

Distributed key-value stores consist of several nodes, sometimes located in geographically different locations, that store the data required for applications, often in-memory. Distributed key-value stores, especially when distributed to multiple servers globally, could reduce the latency between the servers and clients by reducing the geographical distance between the client and the closest server.

There are different approaches to scaling data storage systems to deal with scale. The following video (Video 1) introduces you to some basic ideas and principles behind scaling in the context of databases. Some of these techniques also apply to key-value stores.


Video 1: Database Scaling.

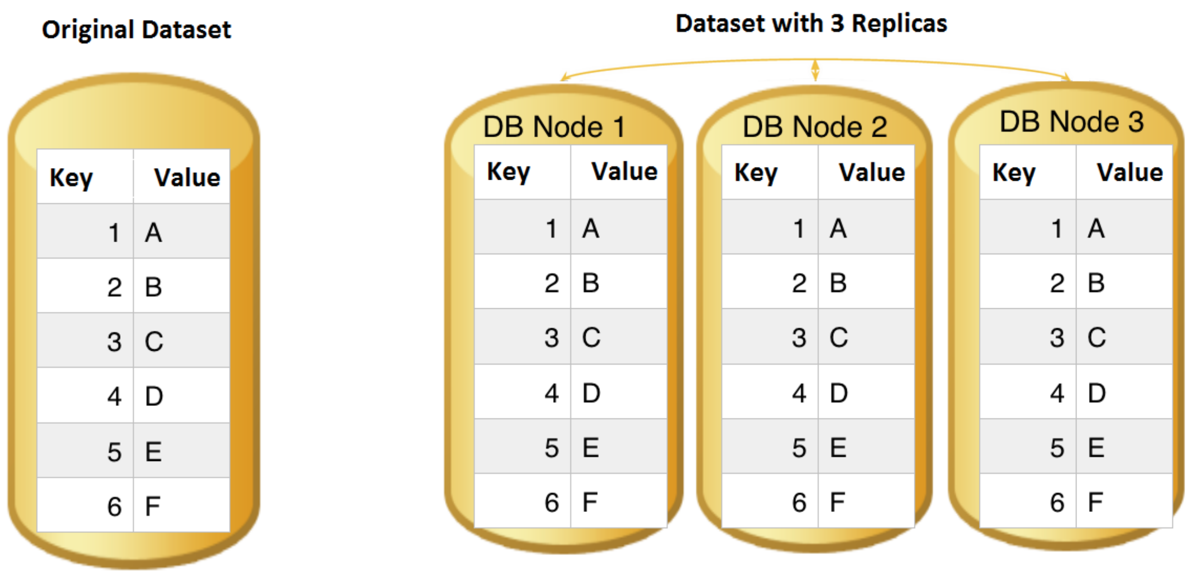
There are different techniques for distributing the data across the available storage servers, each with their own advantages and disadvantages. Some techniques, such as **sharding**, favors high `PUT` performance throughput, since only one node needs to be updated. However, this is at the expense of `GET` performance throughput since an extra step is introduced to find out which shard contains the data that needs to be read. Furthermore, a sharded system is more vulnerable to failures, since the data is distributed across servers, and a failure in any one of the servers can affect availability and may lead to the loss of data. Conversely, techniques like **replication** favor high

`GET` performance, since clients can fetch data from the node closest to their geographical location, or many `GET` requests can be satisfied by a number of servers in a cluster. However, this comes at the cost of `PUT` performance, as each `PUT` request needs to update all replicas associated with that particular key.

In this project, you will be implementing a Coordinator for a distributed key-value storage system, which supports both replication and sharding schemes. After completing this project, you should understand how each scheme works and the scenarios where they apply.

Introduction to Replication

One way to improve performance among distributed key-value stores is replication, which simply replicates the storage on many hosts so that a read query to the storage can be satisfied by any one of the replicas of the datastore. The replicas can be in the same data center or spread geographically to reduce the latency of clients in various locations globally. In case of update queries, the request needs to propagate to all replicas in order to preserve consistency across replicas. Replication is particularly important when fault tolerance is of primary importance and also for performance when there exists a record in the database which is accessed very frequently by multiple clients. The image below shows a dataset that has been replicated across three hosts.



Key	Value
1	A
2	B
3	C
4	D
5	E
6	F

Figure 1: A dataset with 3 replicas.

As you will later learn, there are further optimizations that can be made with the different levels of consistency that exists between each replicated storage. However, for now, we will explore the most basic and most stringent consistency level- **strong consistency**.

Strong consistency is one of the stricter consistency levels for replicated distributed key-value stores. It guarantees that at any point in time, all three datastores must have the same value for any given key. Additionally, the order in which the requests arrive is preserved. Thus, while an update is happening for a certain key-value pair, no other updates or reads can occur until the update is complete for all three datastores. Below is an illustration of strong consistency in a system with 3 replicas.

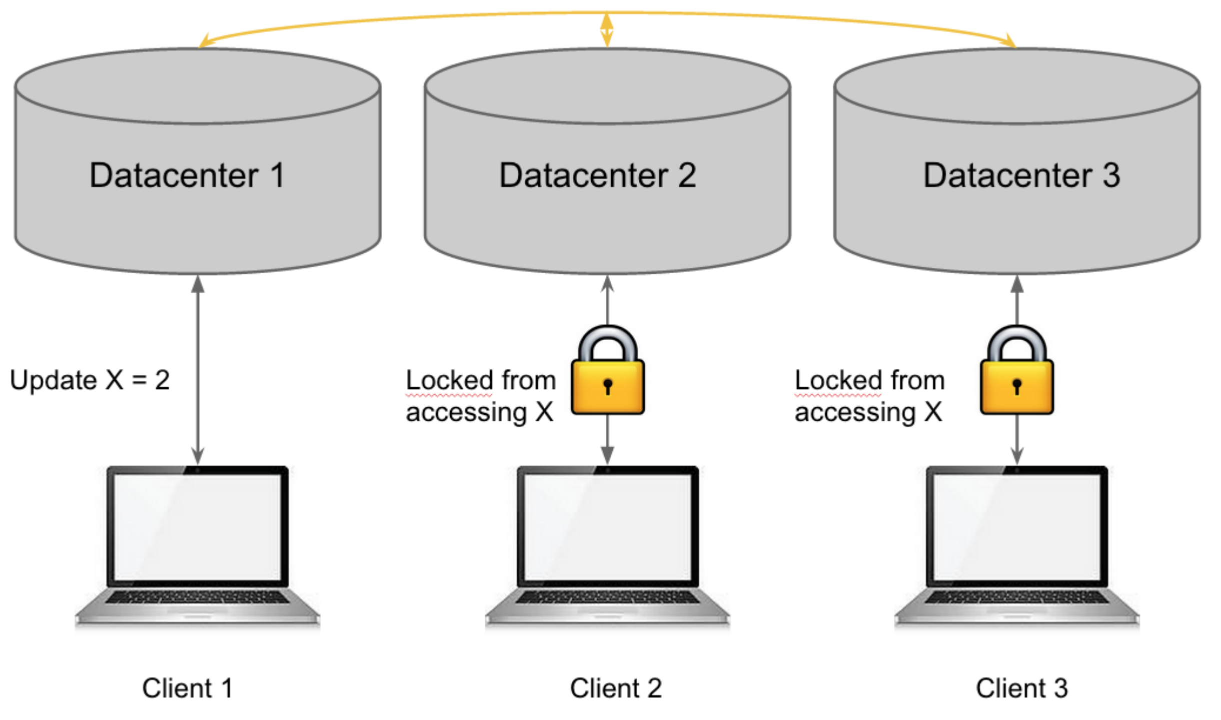


Figure 2: Strong consistency behavior for 3 replicas during an update operation.

Additionally, the order in which the operations arrive will be used as the timestamp for ordering them. To achieve strong consistency, you must make sure that at any point in time, all the clients should read the exact same data from any of the datastore replicas. To summarize, strongly consistent coordinators must abide by these rules:

Property	Explanation

Strict Consistency	At any point in time to the client's perspective, the same key must have the same value across all datastore replicas.
Strict Ordering	The order in which requests arrive at the coordinator must be the order in which they are fulfilled.
Atomic Operations	All requests must be atomic. If one datastore fails to update while the others do, then strong consistency is violated.
Controlled Access	While a write request is being fulfilled for a key, no other requests for that key can be done for any datastore until the pending request is completed. Control for each key should be managed separately, so operations for two different keys should proceed unhindered by each other.

Introduction to Sharding

Recall the concept of sharding, as seen in Video 1, deals with the division and distribution of data among multiple servers. One of the sharding techniques is **horizontal partitioning**, where rows of a storage database (also known as **partitions** or **shards**) are divided among multiple servers. There are numerous advantages to this partitioning approach. In distributed key-value stores that employ sharding, the entire key space is divided among multiple servers, and each machine is responsible for a specific range of keys. This enables a distribution of the data over a large number of machines, potentially GET requests to be processed in parallel. Of course, this assumes that the data is distributed uniformly and the requests for individual keys are also distributed uniformly over the entire key-space.

To summarize before continuing, below is an illustration of a single non-distributed key-value store, a distributed key-value store with sharding, and a distributed key-value store with replication.

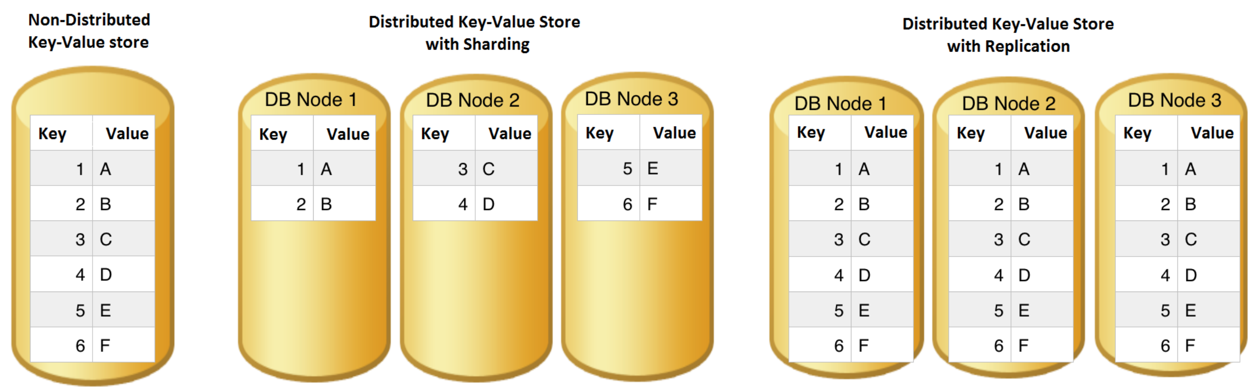

Non-Distributed Key-Value store
Distributed Key-Value Store with Sharding
Distributed Key-Value Store with Replication
DB Node 1
DB Node 2
DB Node 3
Key
Value
1 A
2 B
3 C
4 D
5 E
6 F

Figure 3: Layout of a Non-distributed K-V store, a Sharded K-V store, and a Replicated K-V store.

A key design decision for a distributed key-value store is the distribution of keys over the set of data stores. In this project, you will be implementing a **consistent hashing** algorithm, which will determine the assignment of keys to datastores. A consistent hashing algorithm must always abide by one simple rule:

```
The hashing algorithm must return the same value for the same key at all times.
```

However, while this is the only functional requirement, a consistent hashing algorithm is not practical unless it also distributes the keys fairly across the different shards. As such, a good consistent hashing algorithm will also attempt to distribute the keys evenly. Additionally, a good consistent hashing algorithm must also be able to handle failures, but that is beyond the scope of this project.

The Scenario

Now that you have successfully completed your first rotation, Carnegie SoShall would like to challenge you to further your skill sets with storage in your next rotation. SoShall plans to have music or movie lovers as users from across the globe, and hence, learning how to scale data storage systems will be a critical element in SoShall's social network.

One of SoShall's aspirations is to build its own online store where it plans to sell music and accessories. In order to support this online store, it has been decided that a fast in-memory distributed key-value store is required. Now that you have proven yourself to be an apt cloud programmer, you have been assigned the task to build the distributed in-memory key-value store required for the current scenario.

As you can imagine, the distributed key-value store must support the basic operations of `PUT` and `GET` requests to specific keys in the distributed data store.

SoShall plans to use the key-value store for two types of data:

Sales Records
The sales records keep payment credentials, financial details and other important information about each transaction, so they must kept as safe as possible. If SoShall were to lose these records, they would suffer a large loss! Additionally, the users frequently access their purchase history, so there are plenty of GET operations to this data. Because this dataset is commonly accessed, and also requires failure resistance, SoShall has decided that replication with strong consistency is appropriate.

Anonymous Logs
Contains anonymized logs of page visits and songs listened by the site's users. They are simply a mass dump of user's site history and information. They will only ever be used by data analysts approximately once a week or once a month to crunch and analyze site trends, so random GET requests are rare. Additionally, this data is not critical as it can always be re-built from previous logs. As such, the main concern is being able to handle the very large number of PUT requests. Here, SoShall has realized that a sharding scheme is sufficient for this purpose.

To build the proof-of-concept system, it has been decided that three machines will be allocated for this purpose. Figure 4 illustrates the planned system.

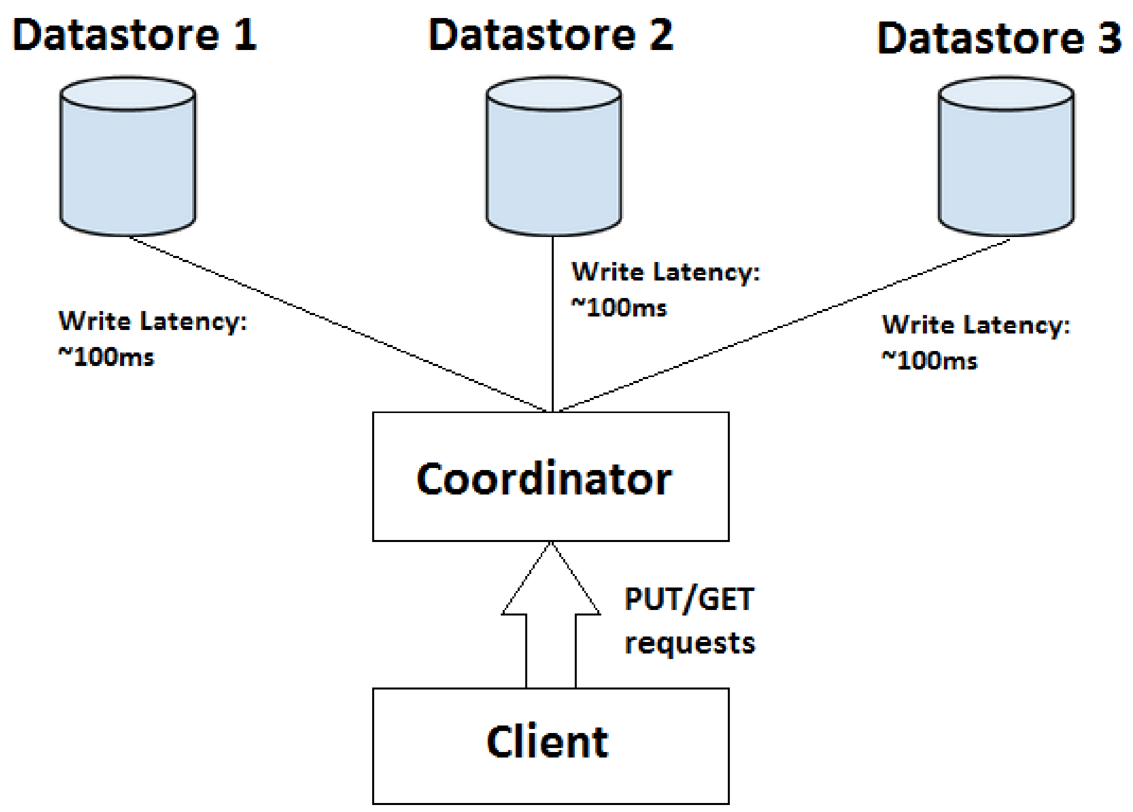


Figure 4: Layout of the Key-Value Storage System.

The individual datastores are simple key-value storage systems that support `GET` and `PUT` operations. The datastore instances have been already configured in the form an AMI for you, so you do not need to modify them. They will launch automatically upon creation.

The only instance you need to implement is the Coordinator, where you will be handling and routing the incoming requests to the appropriate datastores. You will also have to manage the operations by interacting with the underlying datastores using the provided API.

Implementing the Coordinator

The coordinator implementation consists of two parts. In the first part, you will implement replication with strong consistency for the key-value store. Later on, in the second part, you will extend the implementation to also support sharding of the data across the data stores.

Coordinator Requirements

Because SoShall wants the distributed key-value storage system to be as optimized and robust as possible, they have placed a number of requirements on your coordinator implementation. They are as follows:

Property	**Explanation**
Concurrent Execution	Requests must be handled concurrently (multi-threaded) using threads. The skeleton code for this has already been provided to you in the Coordinator instance. If you are unsure about concurrent programming, try exploring the Multithreading Primer.
Deterministic Behavior	Your Coordinator must be free of race conditions, and always have consistent behavior. You will be expected to make use of various concurrency safety techniques at your discretion.
Non-Blocking PUT Operations	Your Coordinator must not block PUT requests from the clients. Rather, it should acknowledge the request and internally prepare to handle the PUT requests whenever the datastores are ready. See Figures 5 and 6 for an illustration of this.
Strict Ordering	For any particular key, your Coordinator must always handle the requests in the order they come. The starter code for making use of the time stamp at which they arrive has been provided. You may make use of this to check the order in which the requests arrive. It is important that ordering is maintained on a per key basis, so the operations for each key must be ordered, but the

	overall operations for all keys may not necessarily be as well.
Non-Caching Coordinator	Your Coordinator may NOT permanently store any Key-Value on the Coordinator instance at any point in time. Temporary storage of Key-value pairs while requests are being processed is fine. Remember, the coordinator should coordinate the operations, not act as a caching storage system.
Dynamic Schema	Your Coordinator should be programmed in a way such that the backend stores can be dynamically changed using the endpoint provided. (Defined later on).

Figure 5 below is an illustration of an incorrect Coordinator which blocks PUT requests:

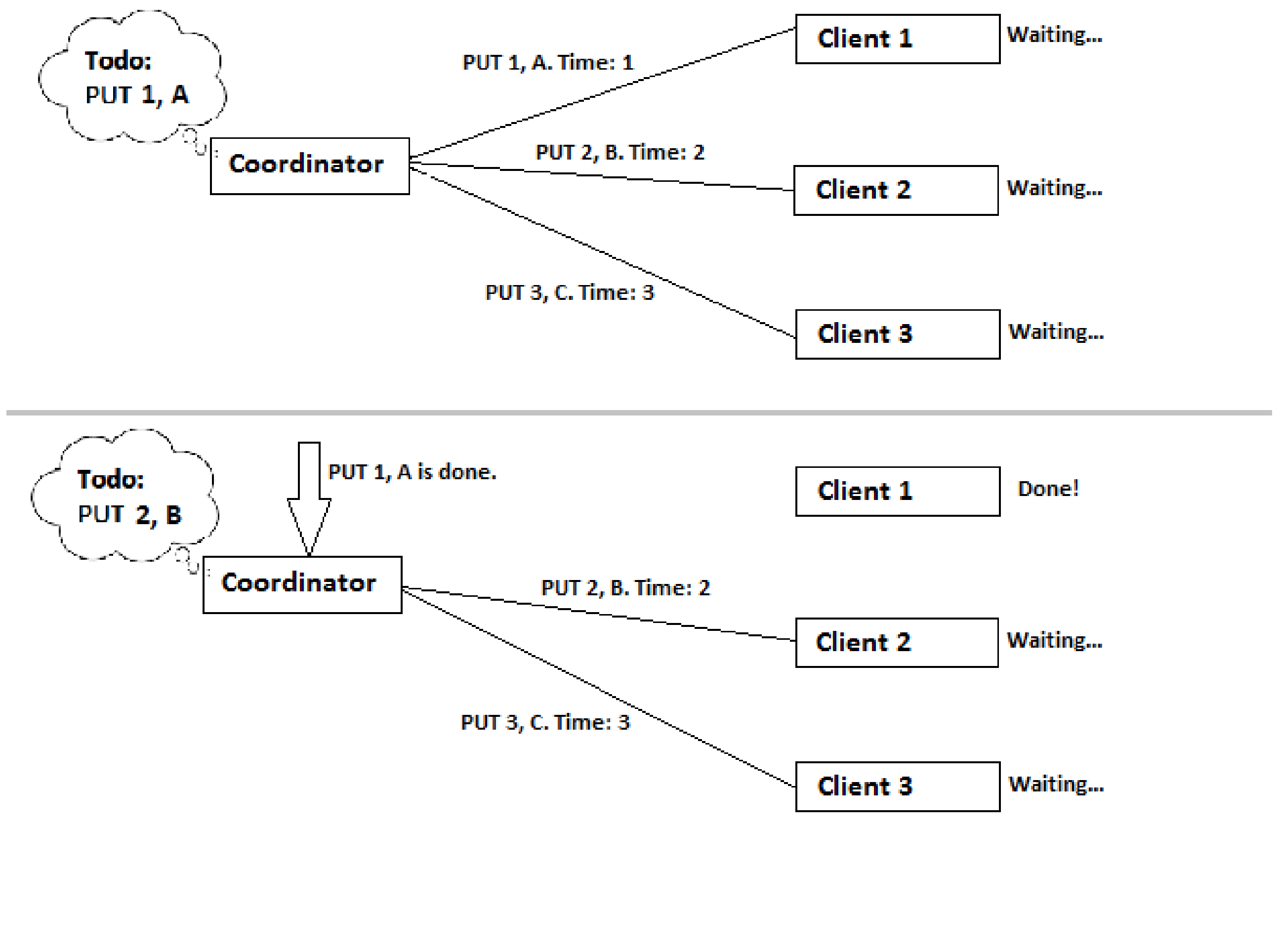


Figure 5: An Incorrect Coordinator blocking incoming PUT requests.

Figure 6 below is an illustration of a correct Coordinator with non-blocking PUT requests:

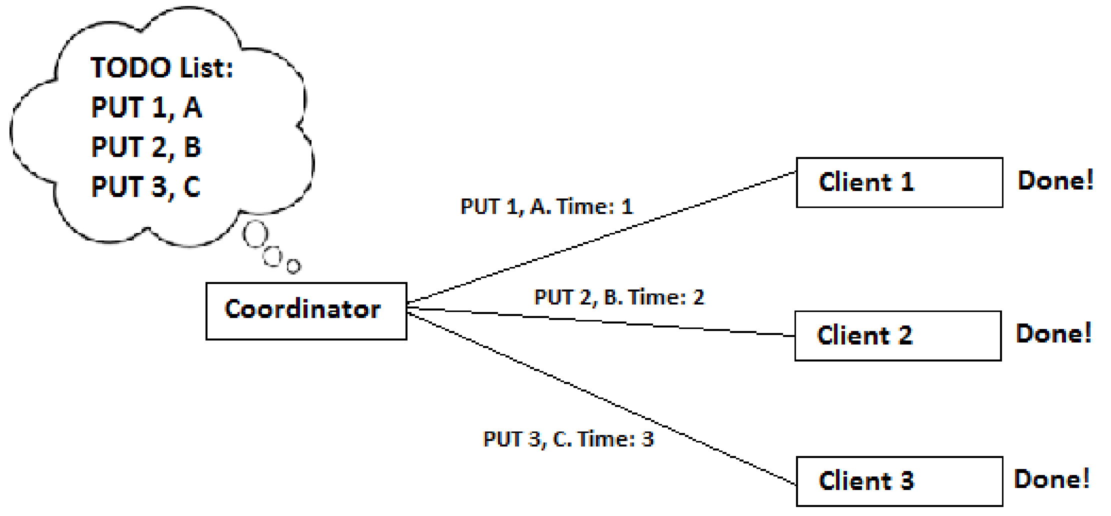


Figure 6: A correct Coordinator with non-blocking PUT requests.

Tasks to Complete

Please read the requirements in this section carefully before starting any instances.
The AMI and tag details are as follows:

AWS Details

The following table contains information regarding various AWS services for this project phase:

Tag Key	**Tag Value**	
Project	3.2	
AMI Name	**AMI ID**	**Instance Type**
P3.2 Datastore	`ami-83ba8ae9`	`t1.micro`

P3.2 Coordinator	`ami-17a4947d`	`t1.micro`
P3.2 Client	`ami-a05d60ca`	`m1.small`

To complete the section, you need to complete the following tasks:

1. Launch **3** datastore instances (`ami-83ba8ae9`) of type `t1.micro` in the **US-East** region. You may need to choose "All generations" in the instance filter when launching to access `t1.micro` instances. The datastore instances contain the code required for the key-value store, and you do not have to program the datastore instance. Make sure your security group allows traffic on port `8080` .
2. Please wait for about 5 minutes for the datastores to start running. You can test if a datastore is running using the test URL `http://[DATASTORE-DNS]:8080/test` from your browser.
3. Launch a coordinator instance (`ami-17a4947d`) of type `t1.micro` .
4. Log on to the coordinator instance. In the folder `/home/ubuntu/Project3_2/vertx/bin/` , you will find a java file, `Coordinator.java` which needs to be filled in to complete the coordinator implementation.
5. To start the coordinator, use the command `./vertx run Coordinator.java` .
6. For the first part, you will be extending the skeleton implementation in `Coordinator.java` to support replication with strong consistency for the key-value store. Please use the described requirements to guide your implementation. The endpoints of the coordinator as well as helper API functions to assist with your implementation are described below.
7. Keep in mind that for the second part, you will extend this implementation to add support for sharding as well.

Endpoints and APIs

The coordinator is a web server (running on Vert.x) having the following external endpoints exposed to the client:

Endpoint	**Description**
`http://[Coordinator-DNS]:8080/storage?storage=TYPE_OF_STORAGE`	This endpoint is used by the auto-grader to specify the type of storage the coordinator has to support. The expected values are replication and sharding. You need to use this value to program your coordinator to handle the expected storage mode.

`http://[Coordinator-DNS]:8080/put?key=KEY&value=VALUE`	This endpoint will receive the key, value pair that needs to be stored in the datastore instances.
`http://[Coordinator-DNS]:8080/get?key=KEY&loc=LOCATION`	This endpoint will receive the key for which the value has to be returned by the coordinator. This request also contains a location parameter which specifies the datastore from which the value has to be fetched by the coordinator. LOCATION is 1 for datastore-1, 2 for datastore-2 and 3 for datastore-3. The coordinator has to return the value associated with the requested key in the specified storage location. For the sharding task, if the given location is not the datastore you are actually storing the value for that key, you should return **0**. If the location parameter is not provided, then you should return the value from the partition that you hash that key to. If the key does not because there has been no PUT request for that key, your server should not crash. Instead, the default response should be "`0`".

In order to help your coordinator interact with the backend storage systems, we have provided a helper class (`KeyValueLib`) with 2 methods that you can access within `Coordinator.java` :

Method	**Description**
`KeyValueLib.PUT(String datastoreDNS, String key, String value)`	This API method will put the value for the specified key in the specified datastore instance.
`KeyValueLib.GET(String datastoreDNS, String key)`	This API method returns the value for the specified key from the specified datastore.

NOTE: None of the API methods described above are synchronized (In other words, any number of threads can call these API methods simultaneously).

Other Assumptions

As you design your Coordinator, you may make the following assumptions:

1. For strong consistency, the latency involved in a GET operation is always negligible (~5 ms) since all datastores are consistent, so GET operations will read from a local datastore instance.
2. The datastore instances are assumed to be fully fault-tolerant, i.e., the datastore instances can be assumed to never fail in any scenario for this project. When implementing locking on the datastore instances, you can assume once the `KeyValueLib.PUT` operation returns, then the operation is complete for that storage instance. This assumption is meant to reduce the complexity of the locking mechanism that you have to implement. As a hint, you do not need to implement the Two-Phase Commit (http://en.wikipedia.org/wiki/Two-phase_commit_protocol) (2PC) mechanism.

Implementing Replication with Strong Consistency

Understanding Strong Consistency

Consistency is very important in distributed applications. We will have a detailed discussion on consistency in the next project (Project 3.3). However, for this project, you will need to understand the concept of **strong consistency**:

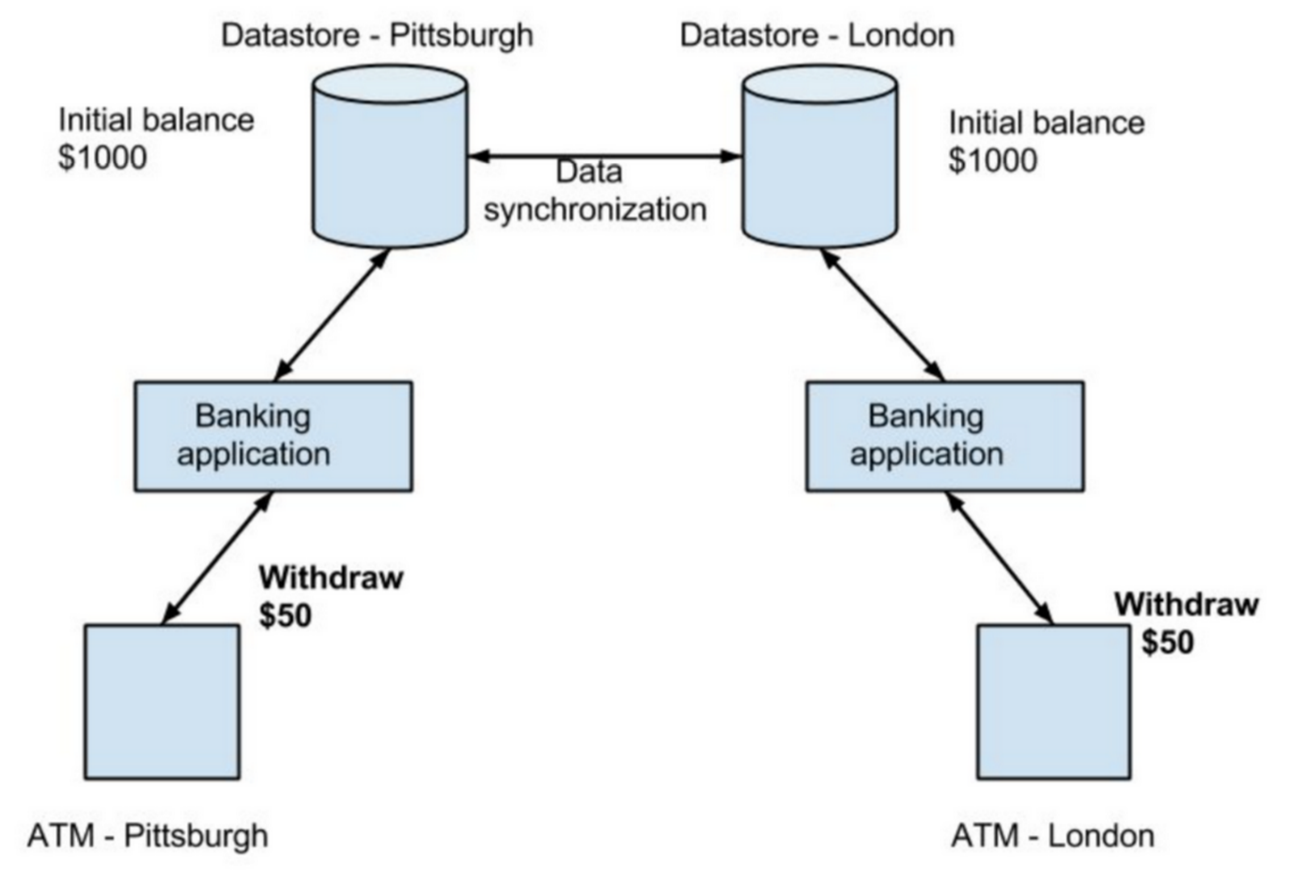


Figure 7: A banking system's datastore.

Let's say that a banking application has its datastores distributed across various locations in order to serve its customers effectively. In this scenario, let's say Joe and his wife Jane hold a shared account in the bank (which currently has $1000 balance). If both of them try to withdraw money from the shared account simultaneously from two different locations as shown in the figure above, there is a possibility for an inconsistent state as the two updates can deduct the amount ($50) twice respectively in the replicas where the requests came, but not $100 overall (balance can $950 instead of $900). This is not an acceptable situation for a banking application. In order to prevent this, the banking application must ensure strong consistency across its datastores as it ensures that only one operation can update all the replicas at a time. If a user is currently trying to update the datastore (by withdrawal or deposit), the banking system should block all other users from updating this specific user's data in all its datastores. The next transcation is allowed to modify the datastore only after all the datastores are made consistent from the previous transaction.

Implementation Requirements

You must implement strong consistency with replication in your coordinator. So, in addition to the general requirements stated in the previous section, your implementation must also follow these requirements for strong consistency. These were also provided to you in the first section of the writeup detailing strongly consistent data stores.

Property	**Explanation**
Strict Consistency	At any point in time to the client's perspective, the same key must have the same value across all datastore replicas
Strict Ordering	The order in which requests for a key arrive at the coordinator must be the order in which they are fulfilled. It is important that ordering is maintained on a per key basis, so the operations for each key must be ordered, but the overall operations for all keys may not necessarily be as well.
Atomic Operations	All requests must be atomic. If one datastore fails to update while the others do, then strong consistency is violated.
Controlled Access	While a write request is being fulfilled for a key, no other requests for that key can be done for any datastore until the pending request is completed. Control for each key should be managed separately, so operations for two different keys should proceed unhindered by each other.

Hints and Suggestions

1. If you are not familiar with concurrent programming, you should explore the Multithreaded Programming in Java primer, which will walk you through multithreading, thread safety, and strategies for maintaining ordering of operations.
2. You need to perform explicit synchronization for ordering the PUT requests. You need to implement a mechanism to lock the datastores before performing a PUT operation on them. Be aware of the situation that two or more requests trying to perform a PUT or GET operation can lead to a race condition (http://stackoverflow.com/questions/34510/what-is-a-race-condition). You need to implement a mechanism to acquire locks for requests before performing the operations (PUT/GET) on the datastores. Your code has to be free of race conditions.
3. There are several ways to handle locks (and race conditions). Java provides a nice way (https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html) to handle synchronization between multiple threads. You may also want to read about synchronization (https://docs.oracle.com/javase/tutorial/essential/concurrency/sync.html) in Java in detail before starting.
4. As shown in the primer, there are a number of data structures at your disposal for maintaining the ordering of operations. Remember, the timestamp is the key for ordering, so an ordered data structure like PriorityQueue (https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html) may be useful to your design.
5. At any point in time, if you want to reset your datastores for testing, you can request the following URL: `http://[Datastore-DNS]:8080/flush`.
6. You will observe a huge delay between SSHing to the Coordinator / Client instances and getting access to the shell. Please **DO NOT** use `CTRL + C` to end the process since we are running some necessary scripts.
7. Start early, particularly if you are not familiar with concurrent programming. Although implementing the coordinator may not take more than 100 lines of code, testing can be tricky due to the difficulty in detecting and fixing race conditions.

How to submit

1. Launch a `m1.small` (`ami-a05d60ca`) client instance.
2. In the client instance, go to the auto-grader folder which is in the base directory located at `/home/ubuntu/` . (This is the directory you arrive at when logging in).
3. The auto-grader folder has the following files - `storage_checker` , `config.prop` , `submitter` and `references` .
4. We provide a storage checker executable called `storage_checker` . In order to use the storage checker, you need to fill the details of the coordinator, and the datacenter instances in the file `config.prop` . (Ensure that the order of the datastore

instances is same in `config.prop` and the `Coordinator.java` , and that the ordering for datastore instances kept the same for when you move on to implement sharding). The `storage_checker` script has several test cases to check your implementation. Once you have updated the details in `config.prop` , you can run the storage_checker using the command `./storage_checker.sh storage_type` . The value of `storage_type` can be either `replication` or `sharding` . So far you only need to test `replication` . The script runs a series of tests and also reports if a test case has succeeded or not. **Before running the storage checker, ensure that the Coordinator and all the datastores are running.**

Implementing Sharding with Even Distribution

Now that you have implemented replication with strong consistency, let's move on to Sharding. First, we must understand the role of consistent hashing in sharding.

Hash Functions in Sharding

A hash function is any algorithm or function that can be used to map data of variable size, called keys, to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. For example, a client's name, which can be of variable length, can be hashed to a single integer. Since usually the codomain of the hashing function is smaller than its domain, collisions (https://en.wikipedia.org/wiki/Collision_(computer_science)) are inevitable (different client names hashing to the same integer). So one of the commonly indicators of the quality of a hashing function is the distributions of the hash values amongst all possible inputs. The more evenly distributed hash values are among the input domain, the better the algorithm.

For example, let us define function A as follows: output 0 if input is 1, 2 or 3; output 1 if input is 4, 5 or 6. Similarly, define function B as: output 0 if input is 1, 2, 3, 4 or 5; output 1 if input is 6. The two functions share the same domain and codomain, but the distribution of the hash value of function A is more even than function B. So, from the performance perspective, function A is better than function B.

In this project, the hashing function is used to decide which data center a data should be stored in. So, the evenness of the data distributed in different data centers depends on the evenness of the hash value distribution. You should aim to design a hash function that distributes the data across the three data centers as evenly as possible.

Task Requirement

When implementing sharding for your Coordinator, you must adhere to the following requirements:

Property	**Explanation**
Consistent Sharding	For any given key, the key-value pair can only be stored in one of the three available datastores. All subsequent updates to that key must be updated in the same datastore. As a result, for any given point in time, any key should only ever exist in one of the three datastores.
Consistent Hashing	You should write a consistent hashing algorithm to determine which of the three datastores each key belongs to. Further specifications are listed below this table.
Individual DC Locking	Requests for individual datastores should not lock other datastores, only requests for that datastore alone. For example, if a PUT request arrives for a key that exists in datastore 1, that should not block a request for a key that exists in datastore 2.
Non-Blocking PUT Operations (DCs)	PUT requests for different datastores should be handled concurrently (multithreaded). For example, if a PUT request arrives for a key that exists in datastore 1, and another request for a key that exists in datastore 2, the two updates should occur concurrently because the keys are in different datastores.
Strict Ordering	Just like with replication, the ordering of requests should be maintained. A GET request for a key should always read the latest PUT value that came before the GET request.

a. For any given key, the hashing algorithm must always be consistent, meaning that it returns the same value for that key.
b. The key “a” must be put into datastore 1 (keep in mind of the ordering of your datastores when updating `config.prop` in the Client instance, you must put datastore 1’s DNS on the line that indicates datastore 1)
c. The key “b” must be put into datastore 2
d. The key “c” must be put into datastore 3
e. All other keys should be distributed as evenly as possible between the three datastores. You will not receive credit for trivial hashing algorithms (such as directing every key to one datastore), however it's not too complex to receive the credit.
f. Algorithms that do a good job of evenly splitting the keys will receive bonus credit.
g. You are NOT allowed to use the `.hashCode()` method of String in JAVA. **Five** points will be deducted from your total score if we find you use it.
h. **Please DO NOT search for others' hash function code from the Internet. Our powerful cheat checking team will identify you and your life will be tougher.**

Hints and Suggestions

1. All the hints in the previous section also apply to this section. Review them if you are stuck. Again, the primer would be helpful if you are having trouble with concurrent programming.
2. For the sharding bonus, you can design a hash function based on a pure mathematical method. Or you can log the keys and examine the keys for patterns. Both ways can lead you to the bonus.
3. Do not forget to clear your data structures when the consistency mode is changed.
4. You can test your implementation for Part 1 (replication) and Part 2 (sharding) separately using the consistency checker. Before the final submission, both parts should be working fully.
5. Once again, **start EARLY** please.

How to submit

To complete the project, after finishing all the tasks above, you can verify your implementation using the auto-grader. Follow the steps given below to complete your submission for this project.

1. SSH to the Client Instance you started in the previous section (or start a new one if you terminated it). In the home directory, change the `config.prop` file accordingly. Use command `./storage_checker.sh sharding` to test the sharding part of your Coordinator program.

2. Once you have verified your implementation using the `storage_checker`, you need to submit the results using the executable `submitter`. You need to copy your implementation of the coordinator `Coordinator.java` to the base directory `/home/ubuntu/` of the Client Instance. Also make sure you add all the references (links and Andrew IDs) in the `references` file in the same folder.
3. Once you are ready to submit your code, execute the submitter by running `./submitter.sh`. The submitter executable runs the storage checker for both replication and sharding storages one after the other, and then it reports the results to the auto-grader server. After running this command, you should be able to see your scores in a few minutes. **Note: Ensure that you place Coordinator.java in the home directory in the client instance before running submitter. Not submitting Coordinator.java can lead to 100% penalty in this project.**
4. NOTE: The `storage_checker` script does not check for all possible scenarios. The TAs will be manually grading your code, specifically looking for code that can lead to race conditions. Ensure that you have thoroughly tested your code for correctness and the requirements mentioned earlier. Also, please be sure to document your code properly to avoid losing style points.

Did I Get This?

Which of the following statements is FALSE?

- ○ A valid Hash function might be adding a random prefix string to the key so that the requests can be evenly distributed.
- ○ PUT operations to the same key should be processed one after another.
- ○ Sharding is suitable in a write-heavy scenario.
- ○ Replication is suitable for a scenario requiring high fault tolerance.

Hint Submit Answer