

Building Cloud Native Applications at Scale with VMware Tanzu GemFire for Kubernetes

Session 1 - GemFire Essentials

Safe Harbor Statement

This presentation contains statements which are intended to outline the general direction of certain of VMware's offerings. It is intended for information purposes only and may not be incorporated into any contract. Any information regarding the pre-release of VMware offerings, future updates or other planned modifications is subject to ongoing evaluation by VMware and is subject to change. All software releases are on an "if and when available" basis and are subject to change. This information is provided without warranty or any kind, express or implied, and is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions regarding VMware's offerings. Any purchasing decisions should only be based on features currently available. The development, release, and timing of any features or functionality described for VMware's offerings in this presentation remain at the sole discretion of VMware. VMware has no obligation to update forward-looking information in this presentation.

This presentation contains statements relating to VMware's expectations, projections, beliefs, and prospects which are "forward-looking statements" and by their nature are uncertain. Words such as "believe," "may," "will," "estimate," "continue," "anticipate," "intend," "expect," "plans," and similar expressions are intended to identify forward-looking statements. Such forward-looking statements are not guarantees of future performance, and you are cautioned not to place undue reliance on these forward-looking statements. Actual results could differ materially from those projected in the forward-looking statements as a result of many factors. All information set forth in this presentation is current as of the date of this presentation. These forward-looking statements are based on current expectations and are subject to uncertainties, risks, assumptions, and changes in condition, significance, value and effect as well as other risks disclosed previously and from time to time by us. Additional information we disclose could cause actual results to vary from expectations. VMware disclaims any obligation to, and does not currently intend to, update any such forward-looking statements, whether written or oral, that may be made from time to time except as required by law.

Agenda

How to use Spring Data and Spring Boot abstractions for Tanzu GemFire

How to build, run, and test your applications at scale

How availability of data can be maintained in light of node/pod failures and/or whole site failures

How developers can use the Tanzu GemFire Operator to quickly instantiate GemFire

How to take advantage of Tanzu GemFire availability to build resilience into your applications

How developers can configure Spring Caching applications to enable Tanzu GemFire for caching based on design patterns widely used in cloud native applications

How developers can accelerate test cycles by implementing applications using mock objects as supported by the Spring Test project

Spring Boot Data Tanzu GemFire Essentials

Application Read/Write into Tanzu GemFire



- **GemFire Native API**

- put, get, putAll(), getAll(), remove()

- **Object Query Language(OQL)**

- Supports querying of nested objects stored in GemFire Region

- **Spring Boot Data GemFire (SBDG)**

- Reduces boilerplate code by annotation support
- Provides enhanced support for unit testing
- Supports Auto Configuration

Spring Framework Integration with Tanzu GemFire



Applies **Spring Framework's** powerful, non-invasive *programming model* in a consistent fashion to simplify configuration and development of **data microservices**

Provides abstraction for GemFire applications

Spring Ecosystem Integration...



Spring Cache Abstraction / Transaction Management



Spring Data Commons + REST



Spring Integration (Inbound/Outbound Channel Adapters)



Spring Cloud Data Flow(SCDF) Sources & Sinks

Spring Framework Integration with Tanzu GemFire



Spring Caching with GemFire

- JSR-107 support using @Cachable annotation

Spring Session support with GemFire

- Transparently offload spring session into cache

SBDG Annotations for Configuring Tanzu GemFire



```
@SpringBootApplication  
@EnableClusterAware  
@EnableEntityDefinedRegions(basePackageClasses =  
Customer.class)
```

```
public class CustomerServiceApplication {  
}
```


SBDG Annotation for Tanzu GemFire Regions



```
@Region("Customers")
```

```
public class Customer {  
}
```

SBDG will create this region in memory or with in gemfire cluster depending on the **@ClusterAware** annotation and its detection of our topology

SBDG Support for Tanzu GemFire Repositories



public interface

CustomerRepository **extends**

CrudRepository<Customer, Long> {

Customer findByNameLike(String
name);

}

Inherits:

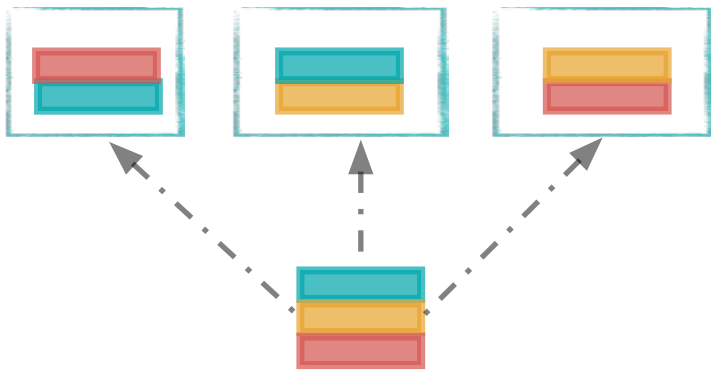
- count()
- delete(T)
- deleteAll()
- deleteAll(Iterable<? extends T>)
- deleteById(ID)
- existsById(ID)
- findAll()
- findAllById(Iterable<ID>)
- findById(ID)
- save(S)
- saveAll(Iterable<S>)

Tanzu GemFire Regions

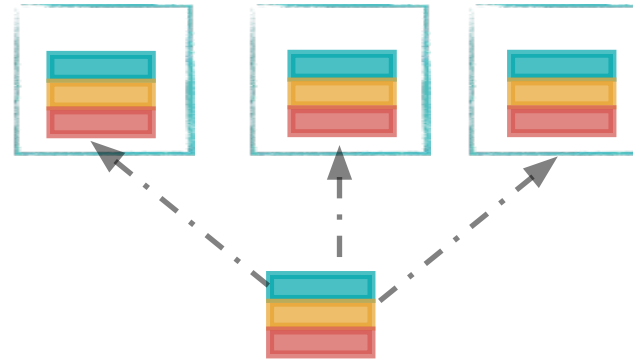
Region Types



Partitioned



Replicated



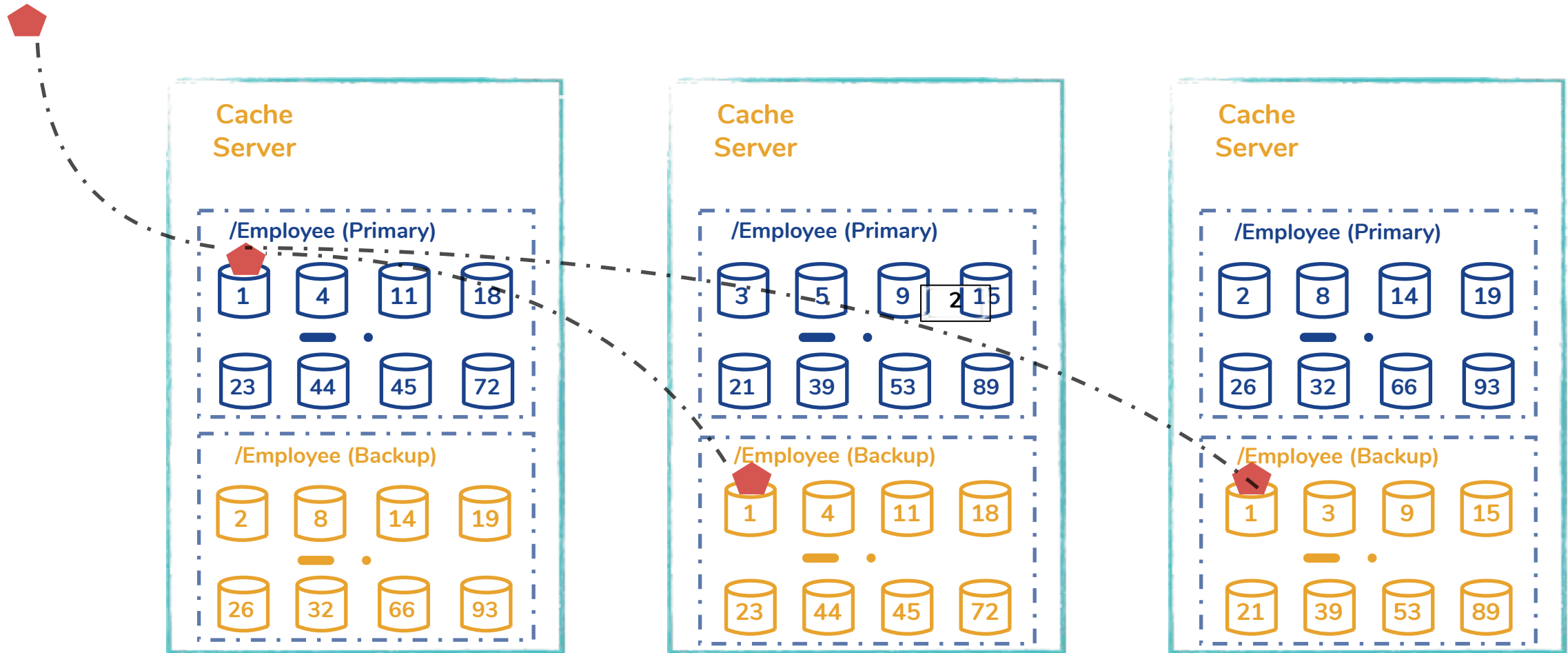
- **Partitioned** Region is the recommended default. Provides horizontal scale *and* replication
- For read/write/update-heavy workloads use **Partitioned** region
- For Many-to-many data relationships use **Replicated** Region
- For read heavy, low update/write data use **Replicated** region



Tanzu GemFire Partitioned Regions

Partitioned region type details

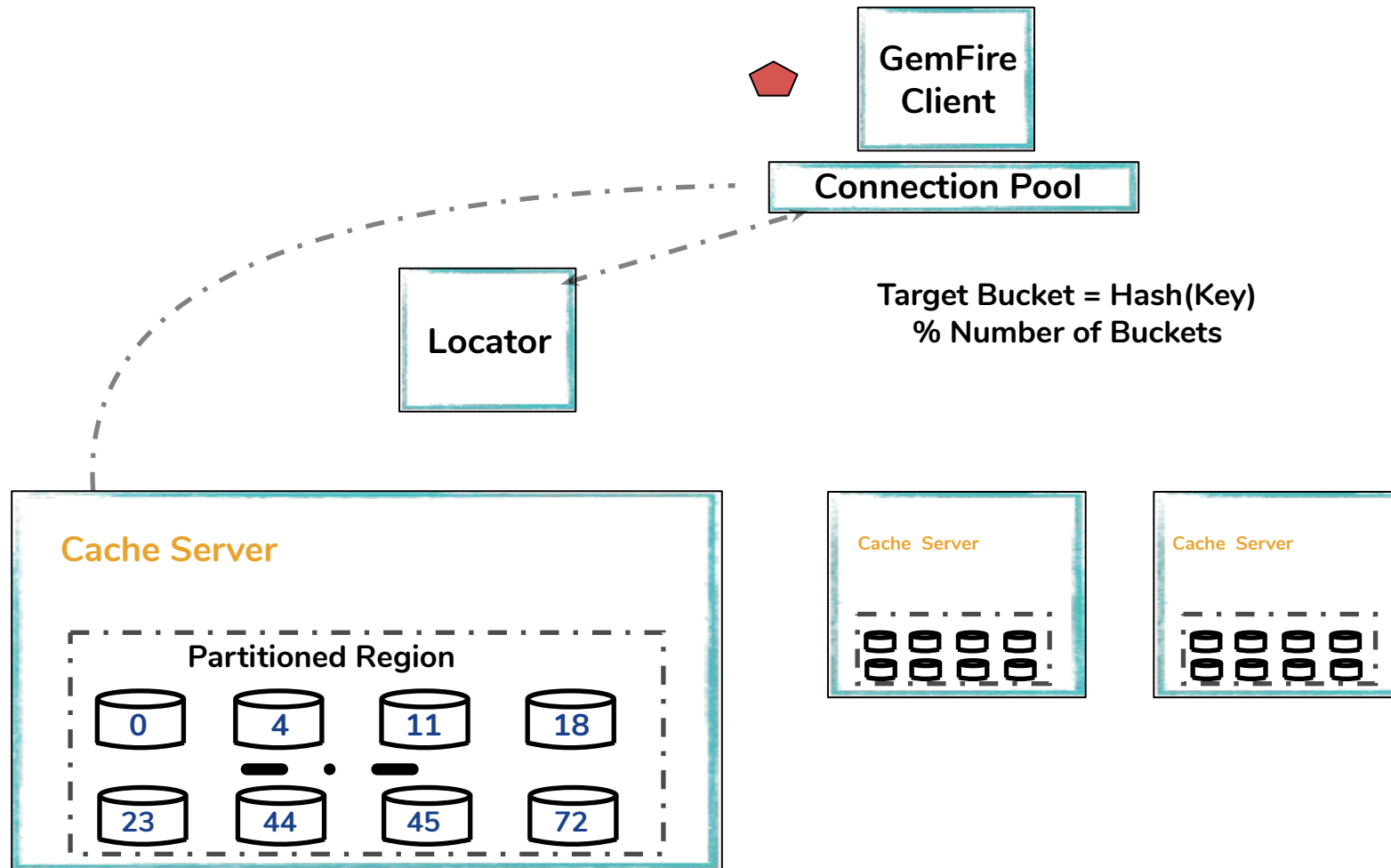
Redundant Copies = 2



Client gets an ACK only when replication to all active backup members is completed

Tanzu GemFire Clients

Clients that are data aware



Target Bucket = Hash(Key)
% Number of Buckets

Total 113 Buckets (Configurable)

- Tanzu GemFire client application pool is aware of where the data is stored on the each server.
- Apps can make direct connection to hosting server to put/get data resulting in fast transactions.

Workshop Strigo Env

- VSCode Installed
- Spring ToolSuite 4 installed
- mvn and gradle
- local gemfire gfsh (gemfire shell)
- Kind for Kubernetes env
- Src code for the labs
 - spring-geode-workshop
 - /labs
 - /decks

<https://github.com/wlund-pivotal/spring-geode-workshop>

Lab 1



SBDG Auto Configuration

What is VMware Tanzu GemFire?

When is it Needed?

Why VMware Tanzu GemFire?

“Every modern application needs a cache” 10x-100x faster than a DB



VMware Tanzu GemFire

The BEST way to store the state for your 12 factor apps

Your app remains stateless

- Gives the appearance of a rich, stateful app. State is stored in a highly resilient, distributed in-memory database

Cache and App are independently scalable as needed

- App will scale utilizing mainly CPU as more concurrent users hit it
- Cache will consume more memory as you dynamically scale

Transparently cache your HTTP session state

- Session follows user connection even if it moves to a different server or foundation

Store your application data

- App data is available at memory speeds anywhere the app runs with reliability due disk persistence

What is VMware Tanzu GemFire



Distributed in Memory Key Value
database Optimized for
Microservices Architecture

Customized plan support through
On Demand Service Broker

- ◆ Session Caching
- ◆ Frequent/Fast Changing Data
- ◆ Static or Slow Changing Data
- ◆ Application Data
- ◆ Update-heavy App Caching
- ◆ Publish and Subscribe
- ◆ Server Side Functions
- ◆ Multi-foundation replication



Performance is Key

We know the number one reason for caching is PERFORMANCE

We are continually improving on our already impressive numbers.
For example in the most recent release we achieved...

- >2x Improvement in server side put and get performance
- 10 percent improvement in client/server put and get performance
(Network latency is the biggest part of client/server performance overhead)



Static or Slowly Changing Data

Patterns used by our Customers



- Front Page for a Website
 - A typical landing or homepage will make dozens of calls to several databases in order to get the data needed for the display.
 - Much of that data is relatively static and therefore can be pre-computed overnight and stored in the cache.
- Frequent Lookups
 - In fact, in many cases that same data is used again and again throughout the daily interactions in other parts of the site.
 - For instance, calculating your co-pay for a procedure requires multiple hits to multiple database tables, but nearly all of the data involved is slowly changing, and therefore a great candidate for caching.

What's my Copay?



Application Data - beyond current login session

Patterns used by our Customers



Shopping cart, preferences, recently viewed items

- Resiliency to app server failure
 - You don't want to store this kind of data in the app server because if it dies, you will lose the data.
- Load Balancing for performance
 - You want the shopping cart to be accessible to the customer no matter which app server they come in on.
- High Concurrency and Horizontal Scalability
 - You expect that there will be very high concurrency on the table that is storing shopping carts, potentially thousands of simultaneous users but not on EACH shopping cart.
 - The traditional, non-scalable database may suffer to keep up with those thousands of concurrent reads and writes.
 - **Tanzu GemFire is horizontally scalable**, and so it can be easily scaled out to handle even millions of concurrent accesses per second.

Applications that require strong consistency

Patterns used by our Customers



Update-heavy apps cannot operate correctly without strong consistency

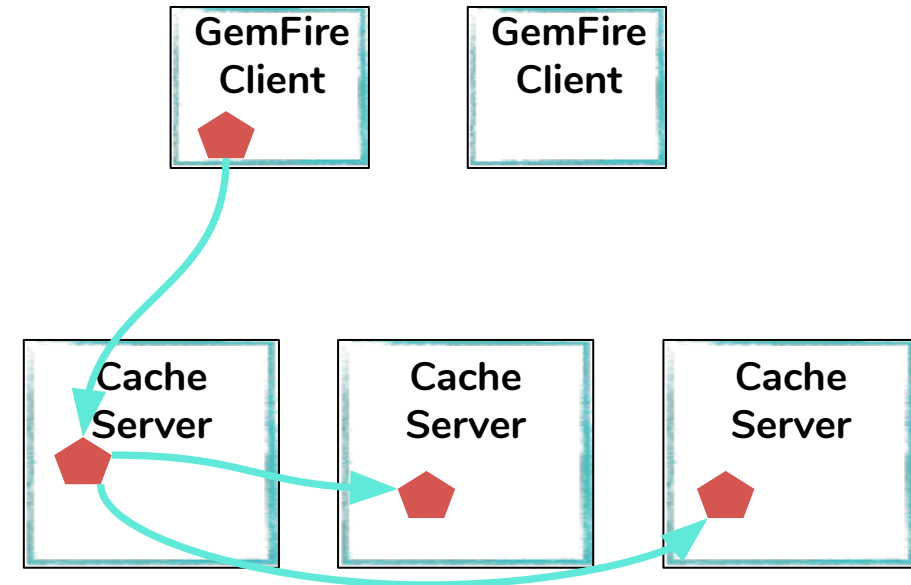
- Typical Enterprise Use Cases
 - Use cases like banking, billing, insurance, inventory, logistics, online e-commerce, manufacturing, risk management and trading are examples of these kinds of apps.
 - Their update heavy nature requires that they operate on correct and up-to-date data. (Think read/modify/write)
- Use-cases for caching highly concurrent updates
 - Require the ability to ensure that the backing data store gets updated in the same order as the cache. Otherwise the cache and the backing data store will disagree in the end.
 - ONLY the in-line caching pattern can properly support highly concurrent update use-cases.



Consistency is a MUST

Our strong consistency is based on synchronous replication

- When a cache client calls put(key, value) replication to backup nodes is synchronous and completes before put returns
 - Writes always happen on Primary for the object



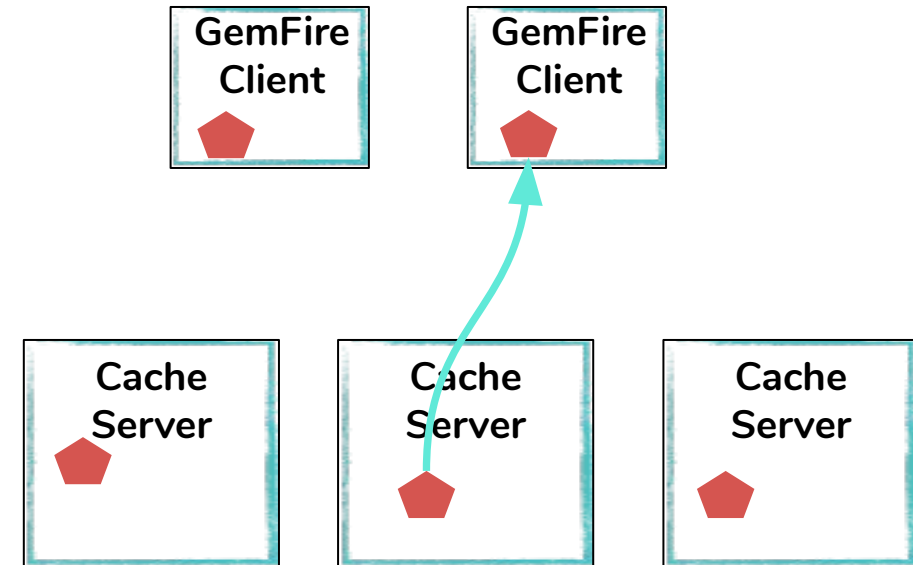
You **ALWAYS** get the **CORRECT** Data



Consistency is a MUST

Our strong consistency is based on synchronous replication

- When a cache client calls put(key, value) replication to backup nodes is synchronous and completes before put returns
 - Writes always happen on Primary for the object
- Because all replication is synchronous reads are guaranteed to get the updated data



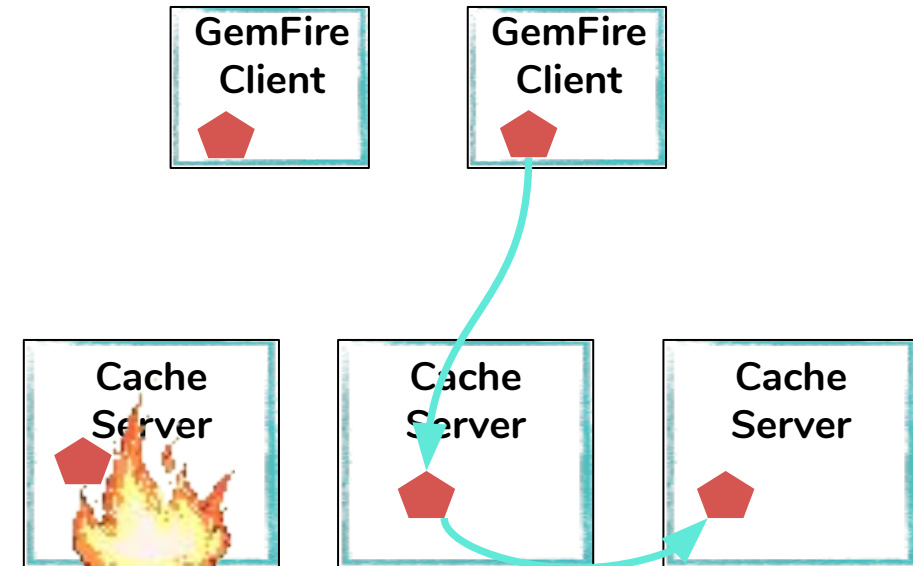
You **ALWAYS** get the **CORRECT** Data



Consistency is a MUST

Our strong consistency is based on synchronous replication

- When a cache client calls put(key, value) replication to backup nodes is synchronous and completes before put returns
 - Writes always happen on Primary for the object
- Because all replication is synchronous reads are guaranteed to get the updated data
- If a Primary fails, a new Primary that is elected already has the updated data



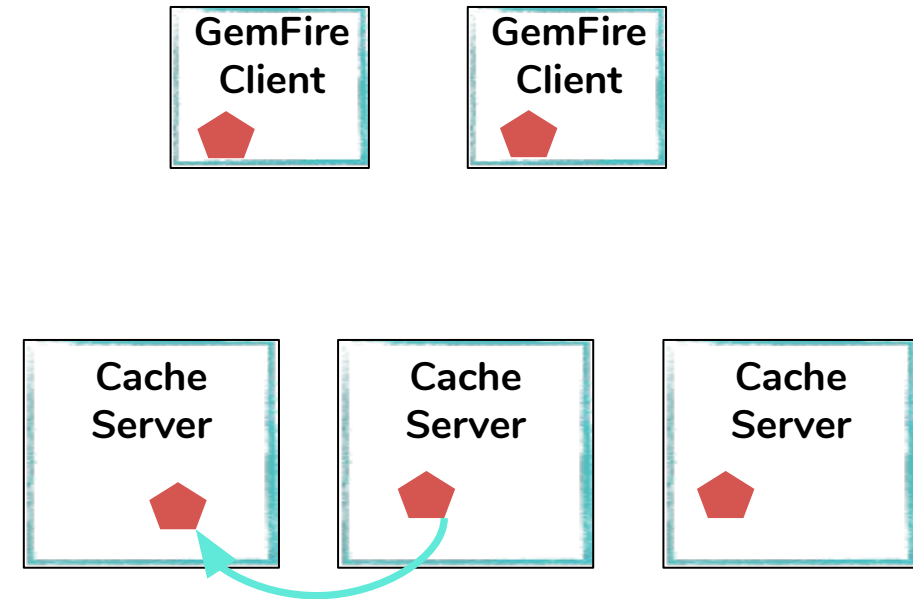
You **ALWAYS** get the **CORRECT** Data



Consistency is a MUST

Our strong consistency is based on synchronous replication

- When a cache client calls put(key, value) replication to backup nodes is synchronous and completes before put returns
 - Writes always happen on Primary for the object
- Because all replication is synchronous reads are guaranteed to get the updated data
- If a Primary fails, a new Primary that is elected already has the updated data
- When a server returns it is re-populated with the latest updates from its peers.



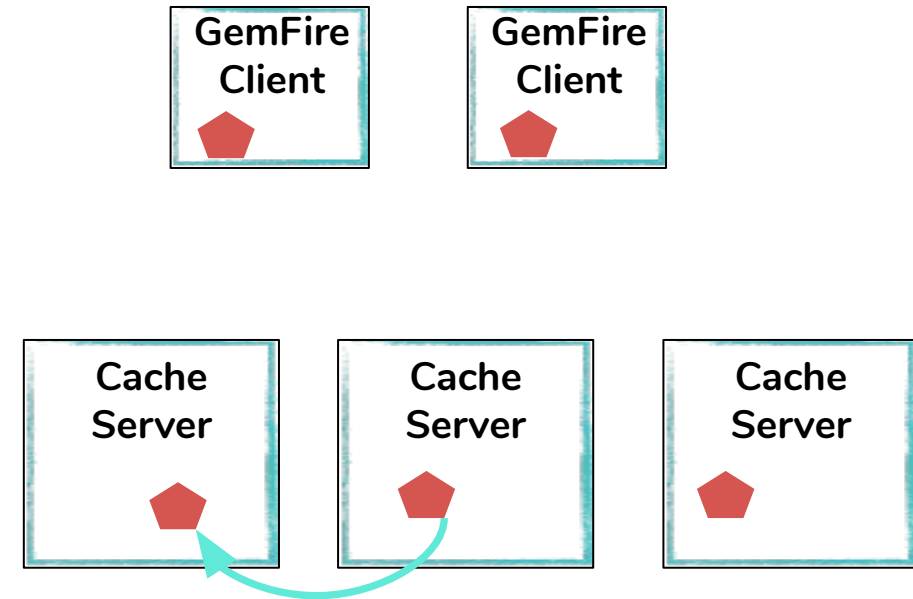
You **ALWAYS** get the **CORRECT** Data



Consistency is a MUST

Our strong consistency is based on synchronous replication

- When a cache client calls put(key, value) replication to backup nodes is synchronous and completes before put returns
 - Writes always happen on Primary for the object
- Because all replication is synchronous reads are guaranteed to get the updated data
- If a Primary fails, a new Primary that is elected already has the updated data
- When a server returns it is re-populated with the latest updates from its peers.



You **ALWAYS** get the **CORRECT** Data

The Value of Tanzu GemFire

Caching is instrumental in microservices architectures



Isolation from shared back-end database

- Bounded Context
- Think of the cache as a materialized view of the back-end data

High Availability

- Keep all state in the cache, let it provide redundancy and high availability
- The service itself is completely stateless while presenting a rich stateful user experience

Performance

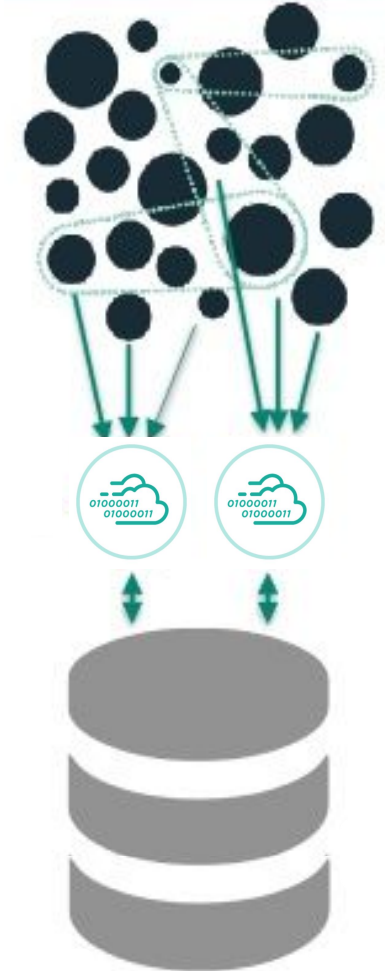
- In-memory performance characteristics for frequently used data

Load Balancing

- No need for sticky sessions
- All instances of service have access to the same cached data

Horizontal scale

- Microservice and data cache can scale independently of one another





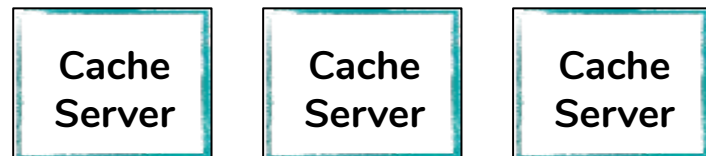
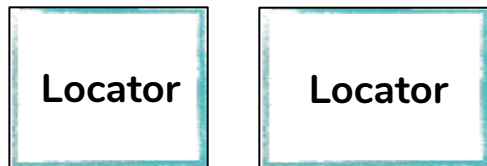
Tanzu GemFire Topology

Tanzu GemFire Members



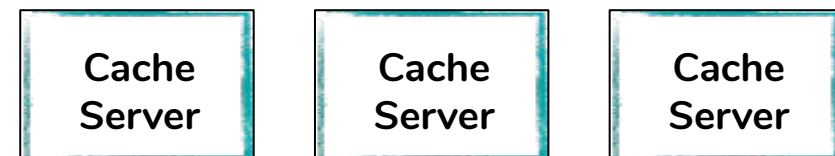
Locator

- Cluster Discovery & Config
- Load Balancing for Servers
- Locators are HA



Cache Server

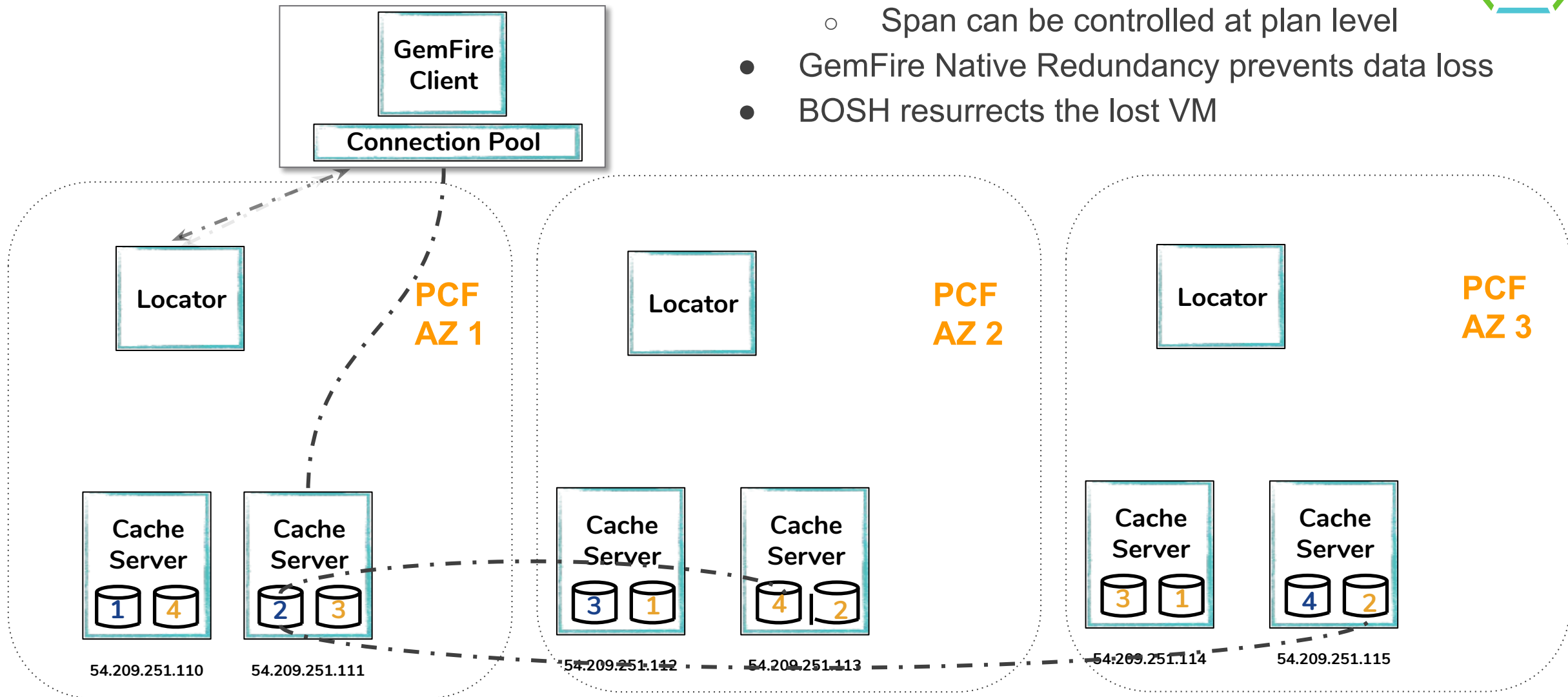
- In-Memory Storage for Data Regions
- Standard Tanzu GemFire Process with one Cache Server per JVM



Tanzu GemFire Topology



- Cluster Spanning Multi PCF AZs
 - Span can be controlled at plan level
- GemFire Native Redundancy prevents data loss
- BOSH resurrects the lost VM

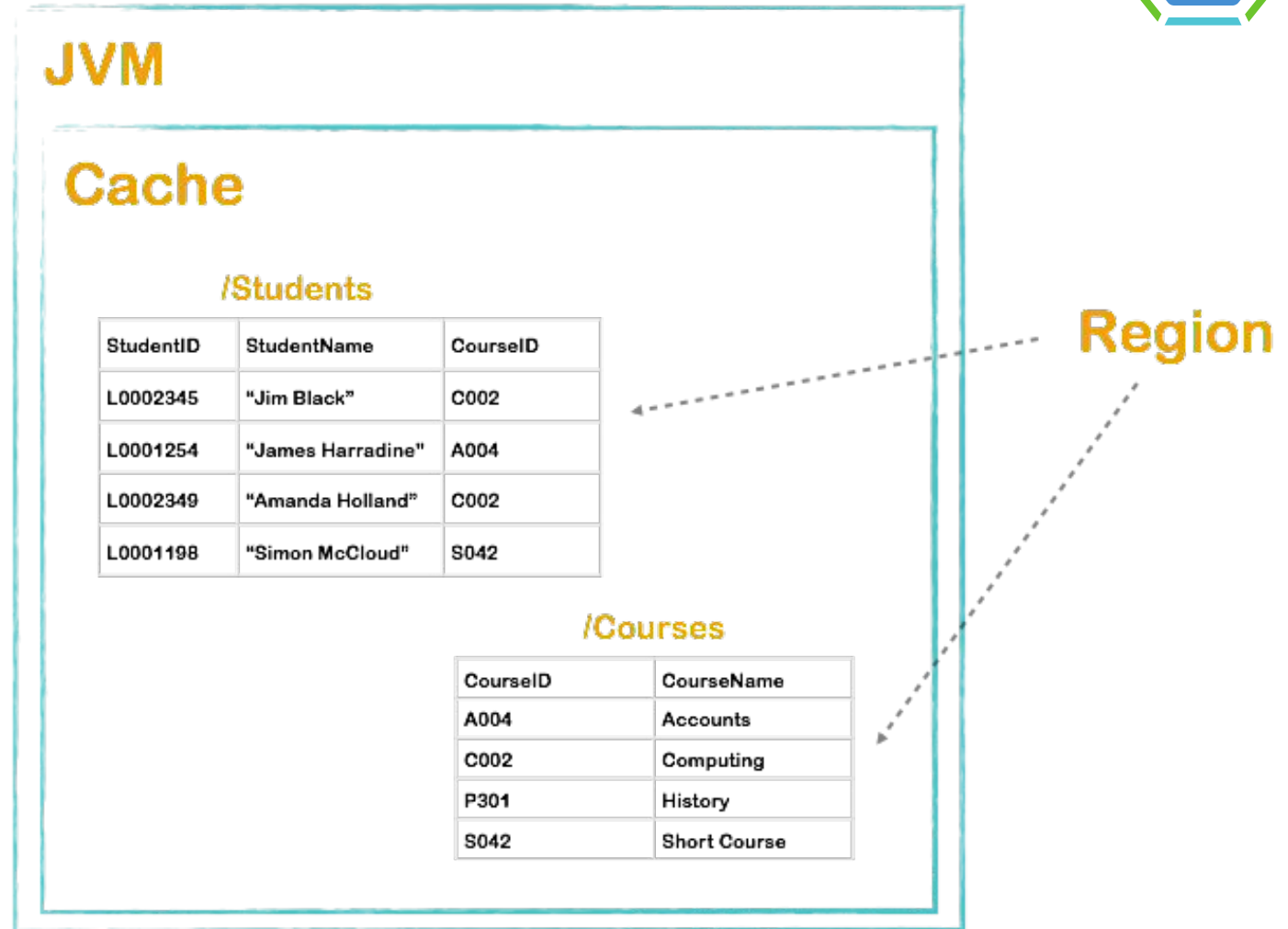




Tanzu GemFire Regions

Storing data in Regions

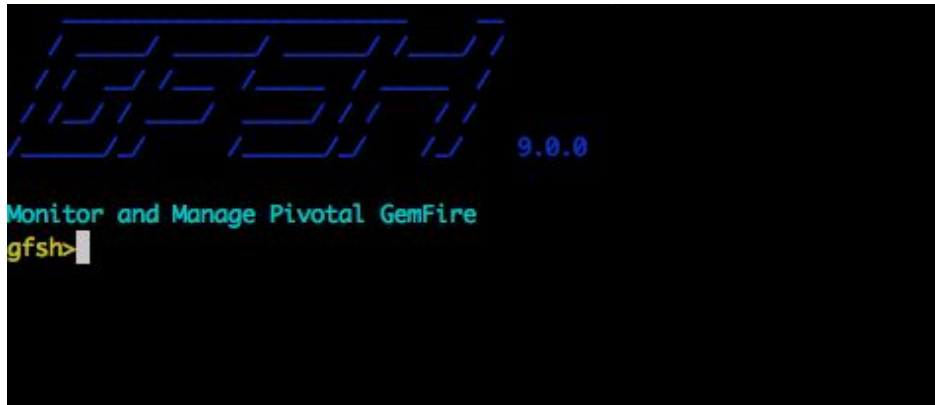
- Synonymous to a Table in NoSQL terminology
- Stores Data in **<Key,Value>** pairs with unique Keys
- Data is sharded across cache members for horizontal scale



Tanzu GemFire Cluster Management

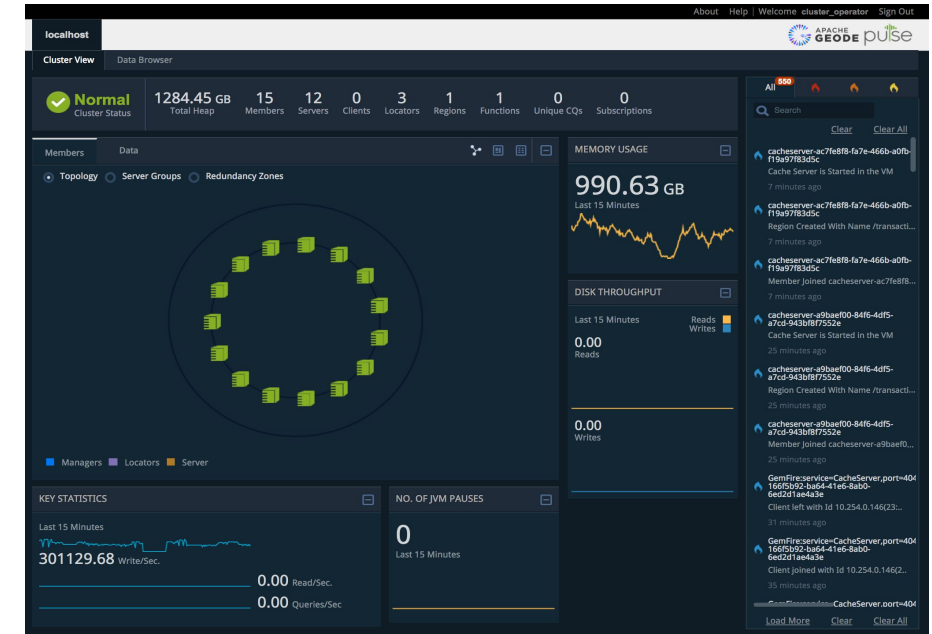


GemFire Shell (GFSH)



- Cluster Administration
- Service Control
- Full Operation Support

PULSE



- Cluster Health Monitor
- Region Query

Tanzu GemFire for Kubernetes

GemFire brings an in-memory key-value store to Kubernetes.



Prerequisites

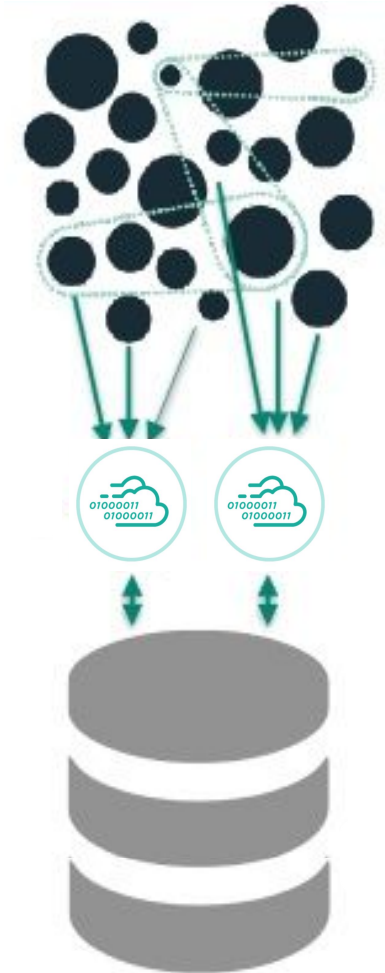
- A Kubernetes cluster, version 1.16 or a more recent version
- kubectl, a version that works with your Kubernetes cluster
- helm, version 3 or a more recent version

Installing Tanzu GemFire


- This Beta version defines a Tanzu GemFire Operator to use when creating a Tanzu GemFire cluster.

Working with Tanzu GemFire Cluster

- Creating gemfire clusters
- Scaling gemfire clusters
- Use an interactive gfsh session.
- Use kubectl to invoke a gfsh command.



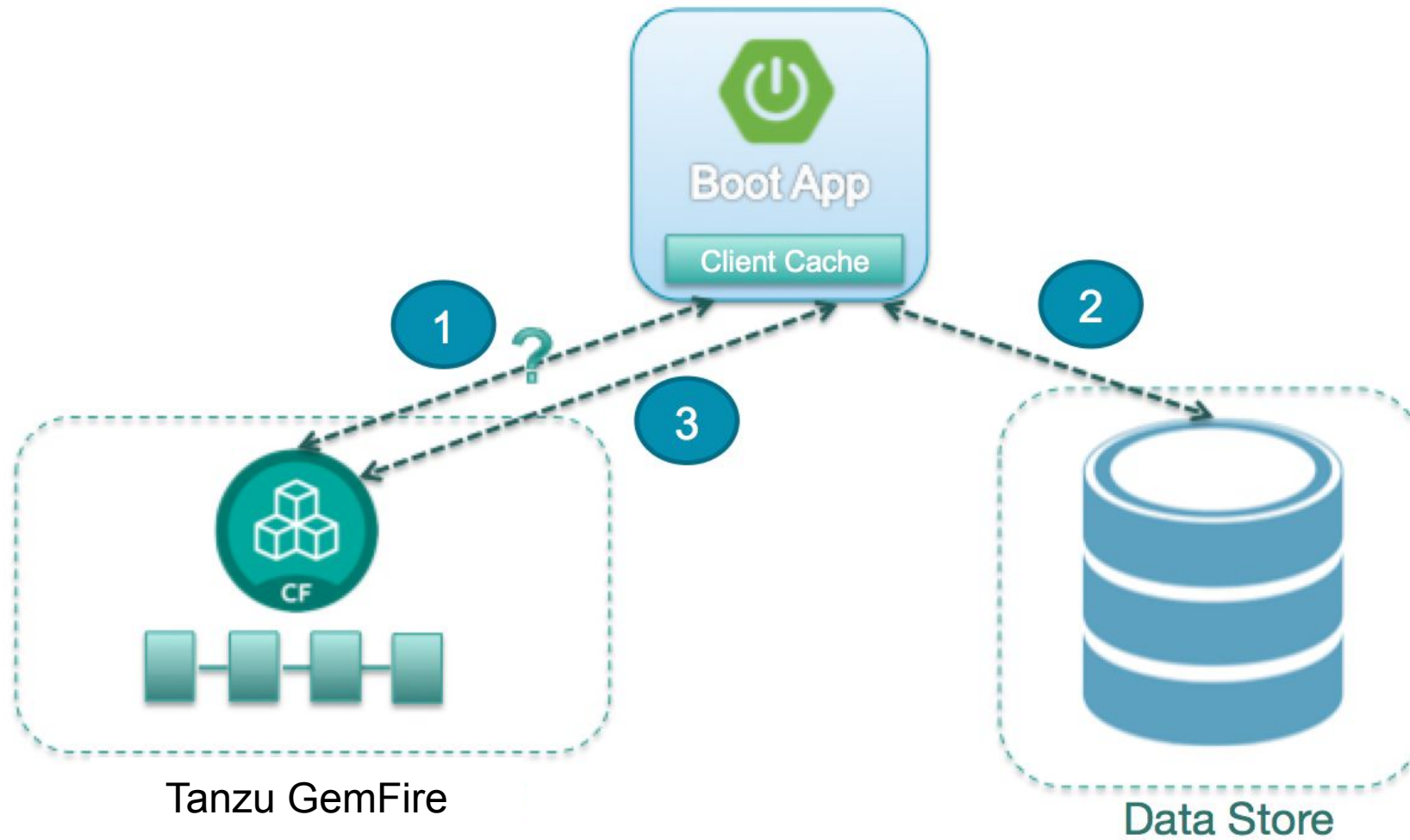
Lab 2



Environment Setup

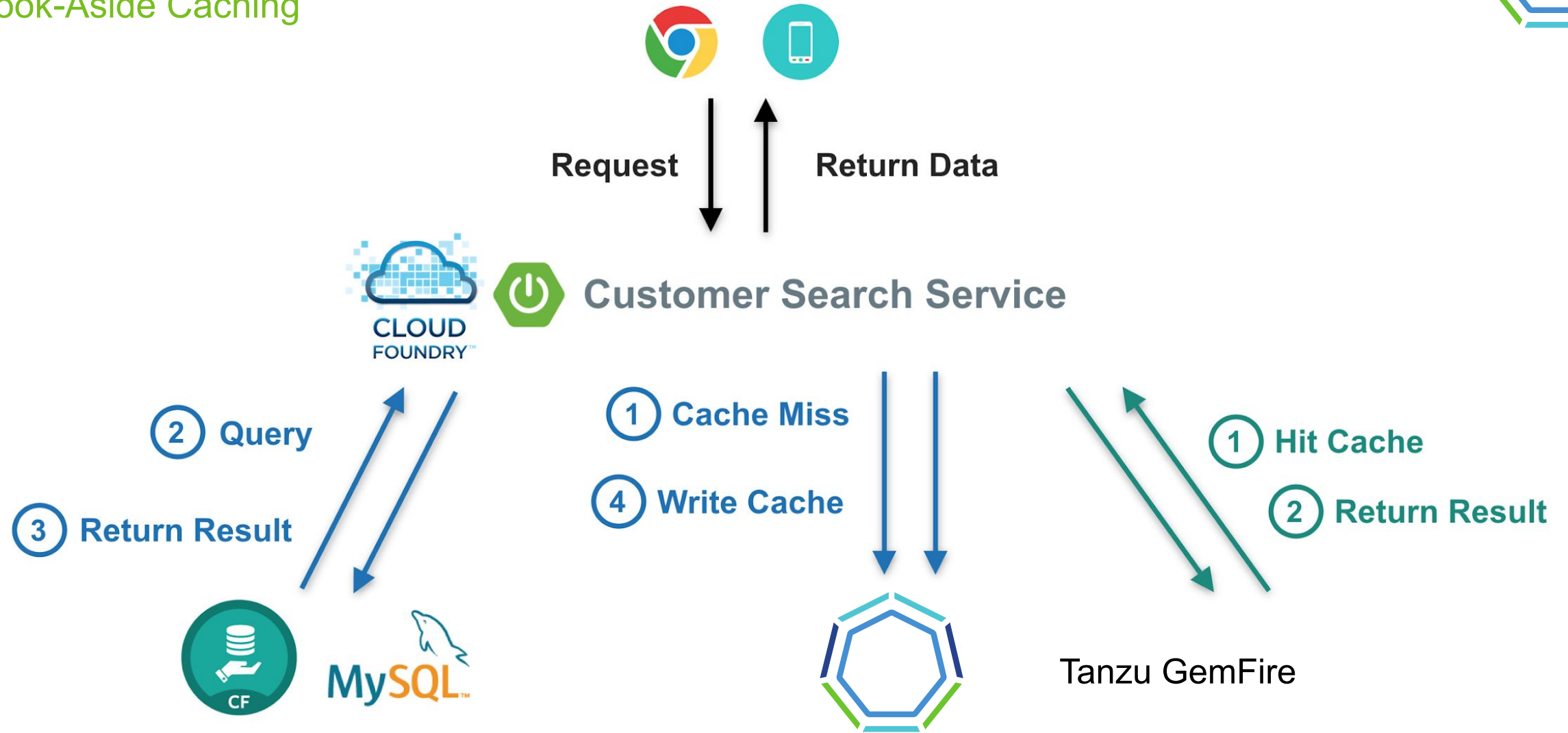
Tanzu GemFire Design Patterns

Design Pattern: Look-Aside Caching



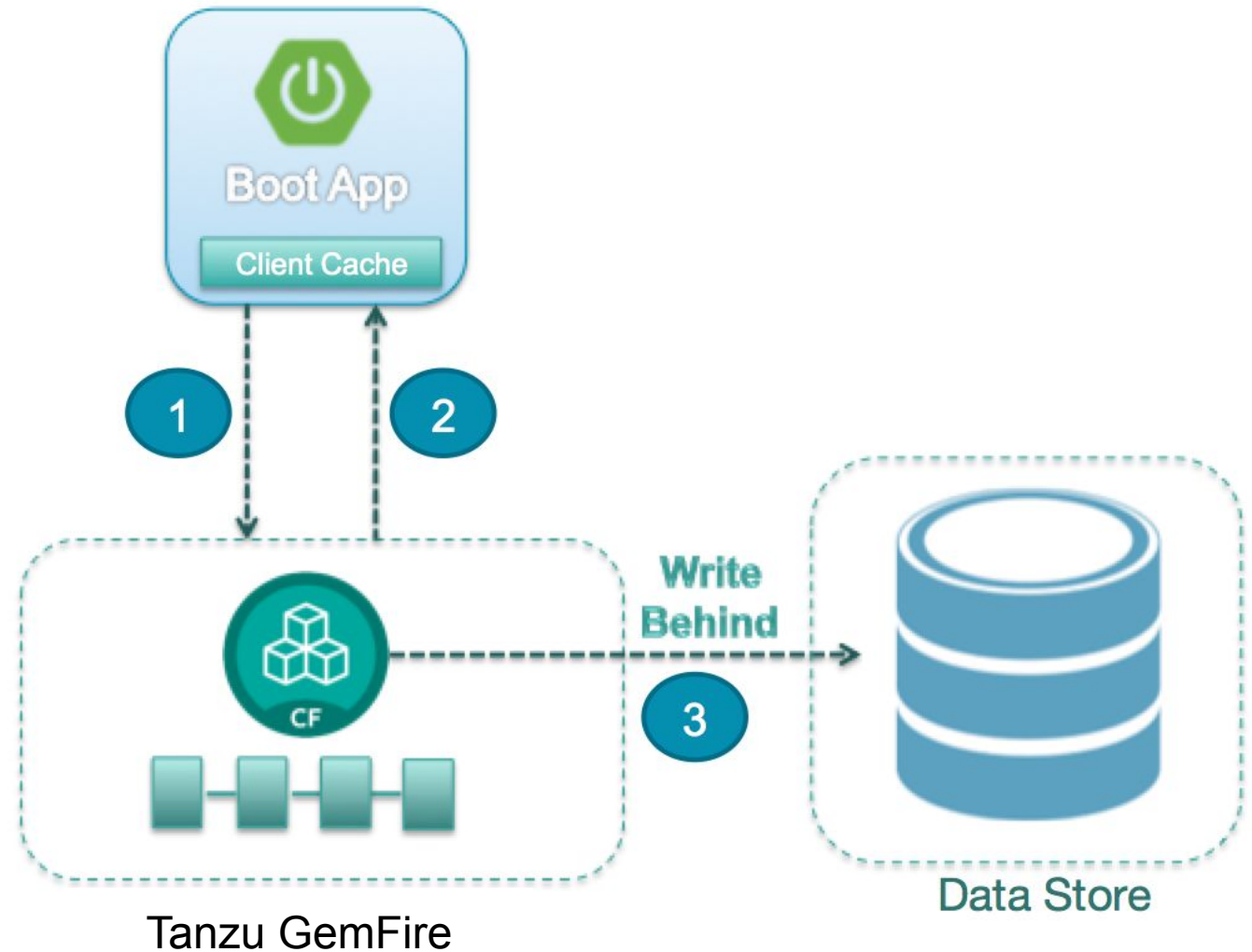
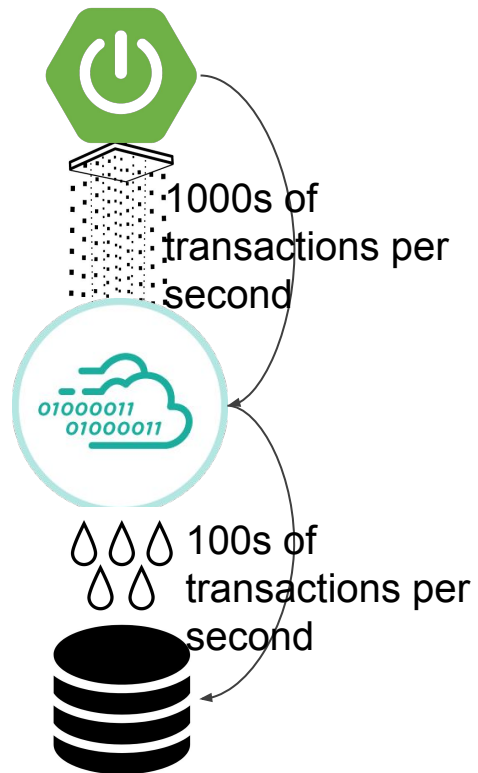
Demo App: Customer Search

Look-Aside Caching



Design Pattern: In-line Caching

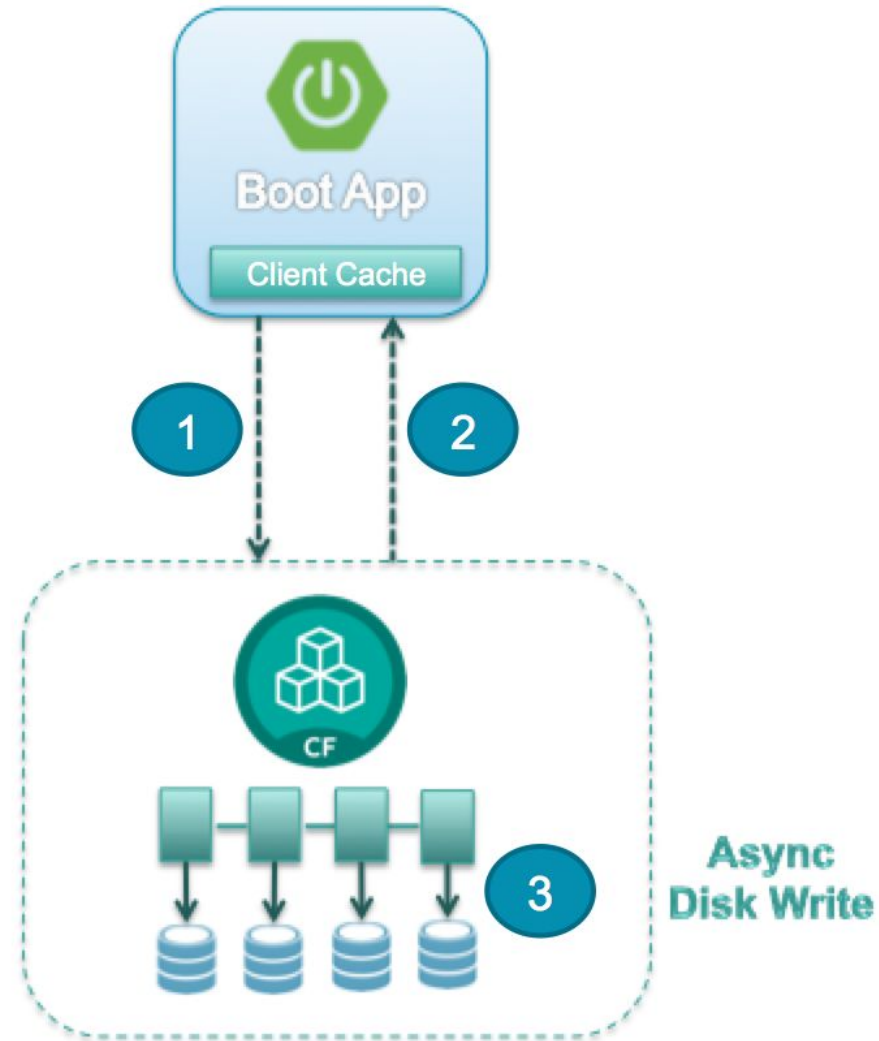
Fast KV Store front-ending RDBMS



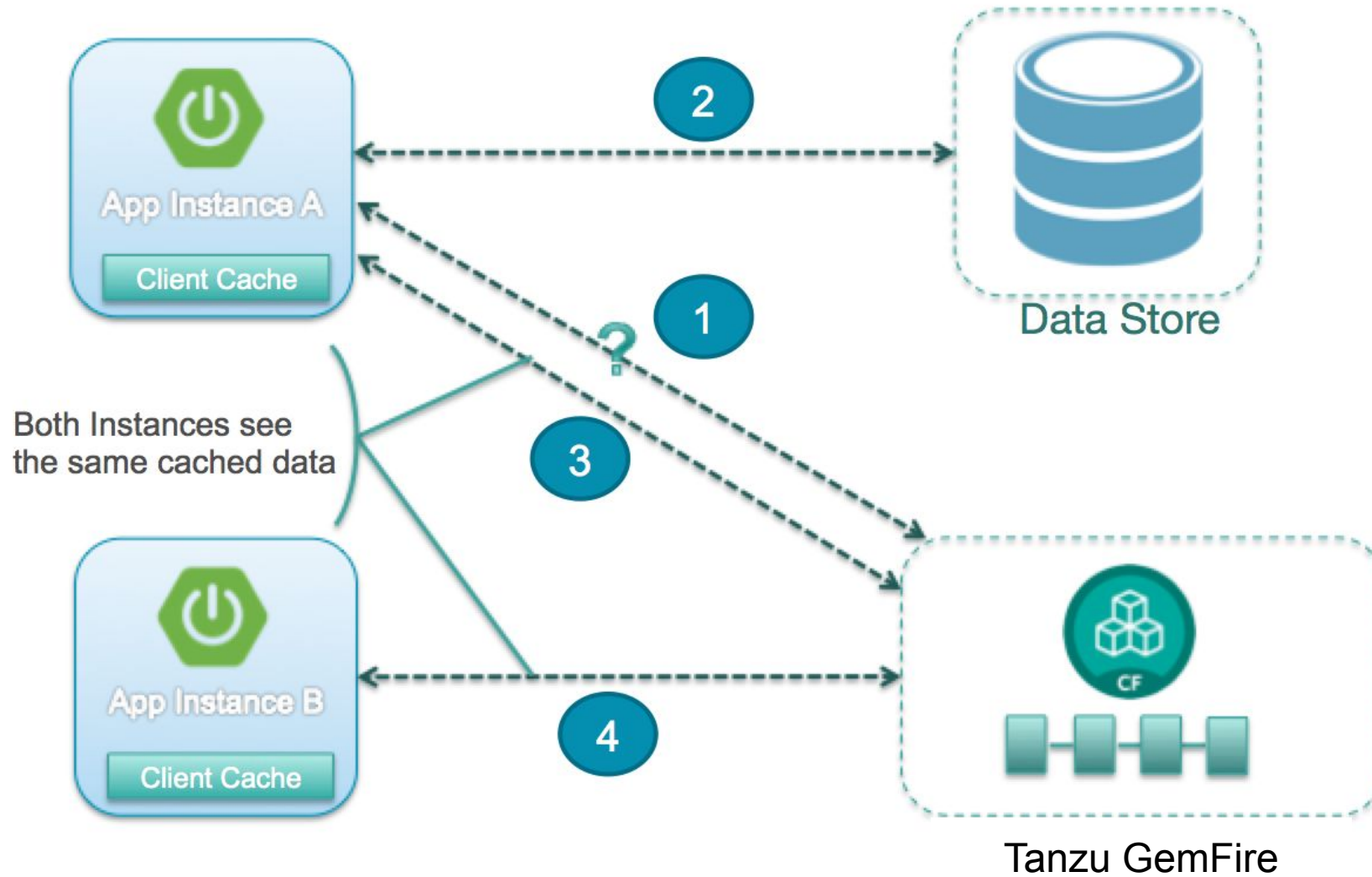


Design Pattern: Disk Persistence

High Availability with Disk Persistence



Design Pattern: Shared Cache





Design Pattern: Client Side Events

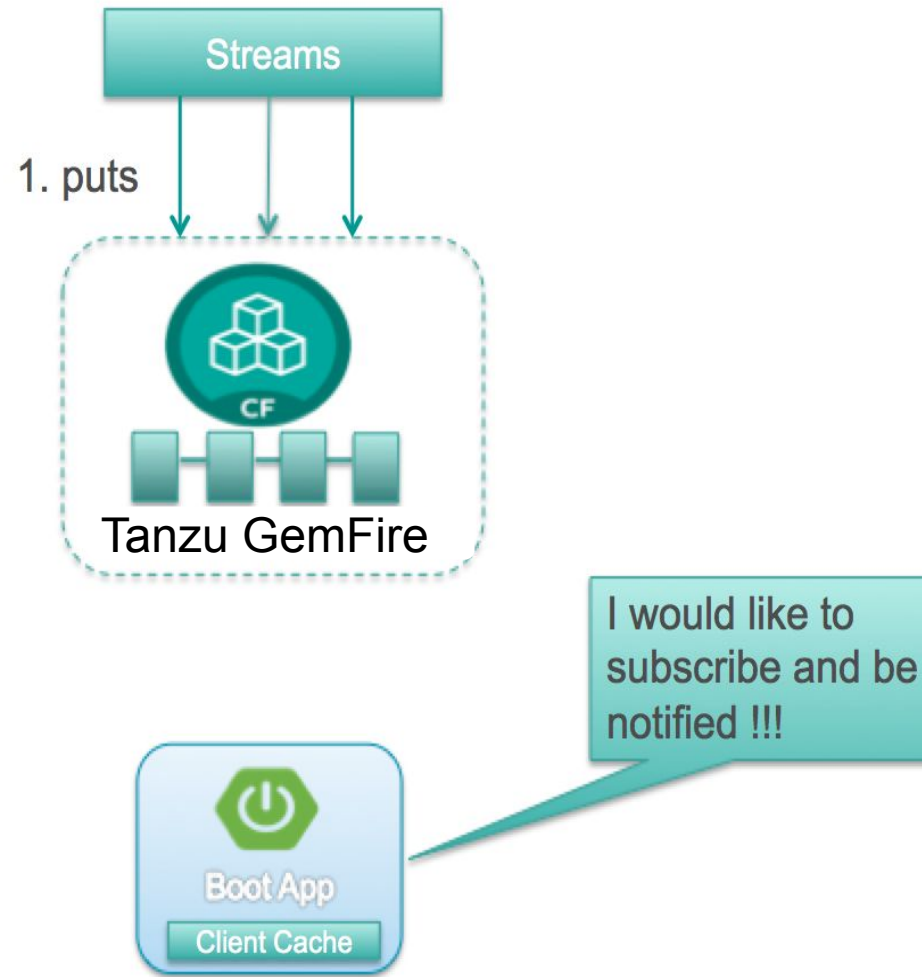
Client Subscriptions

- Client interest in server side event changes
- Filter events based on RegEx Interest

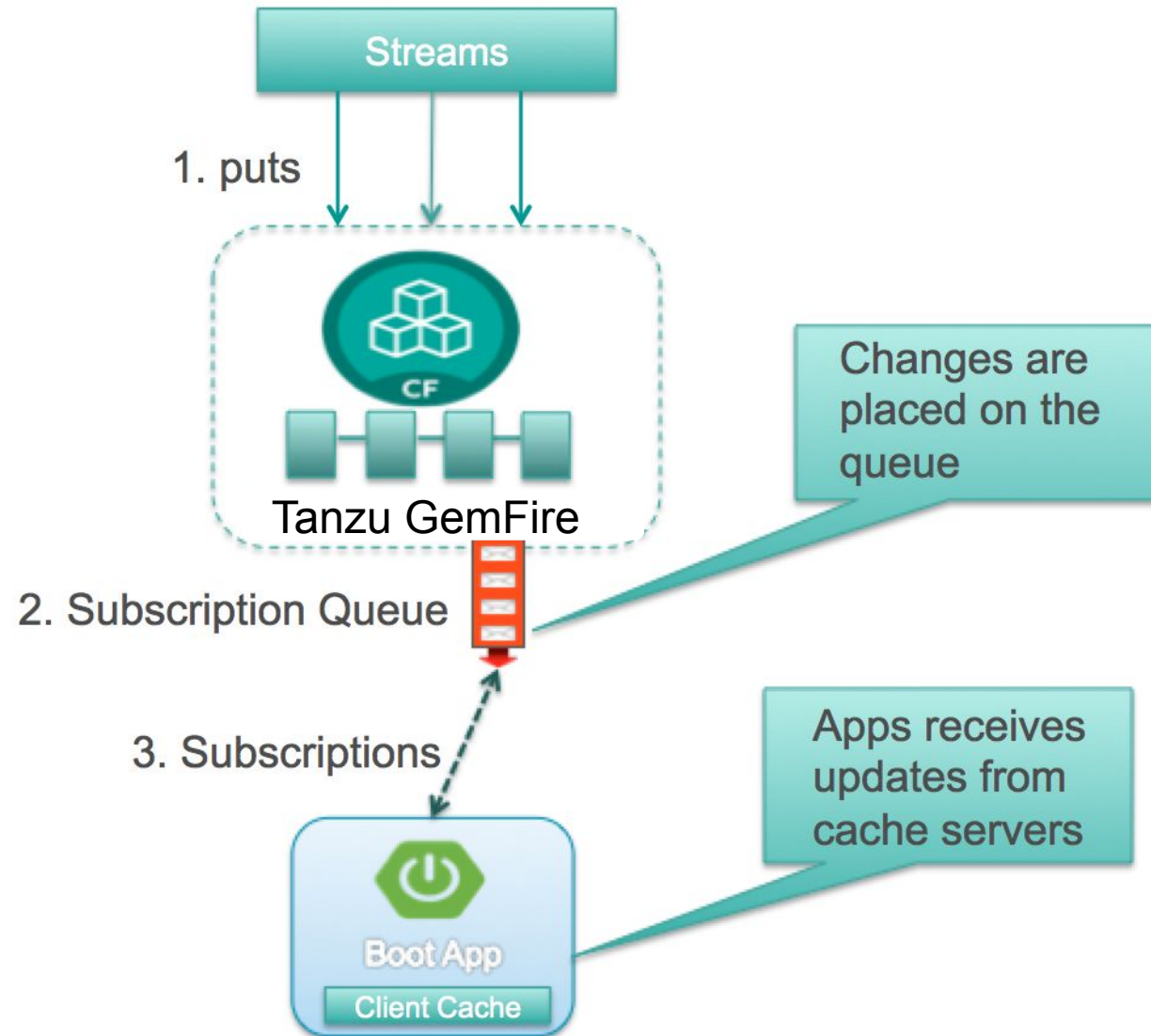
Continuous Query(CQ)

- Register interest through server side queries
- Suitable for creating dashboards

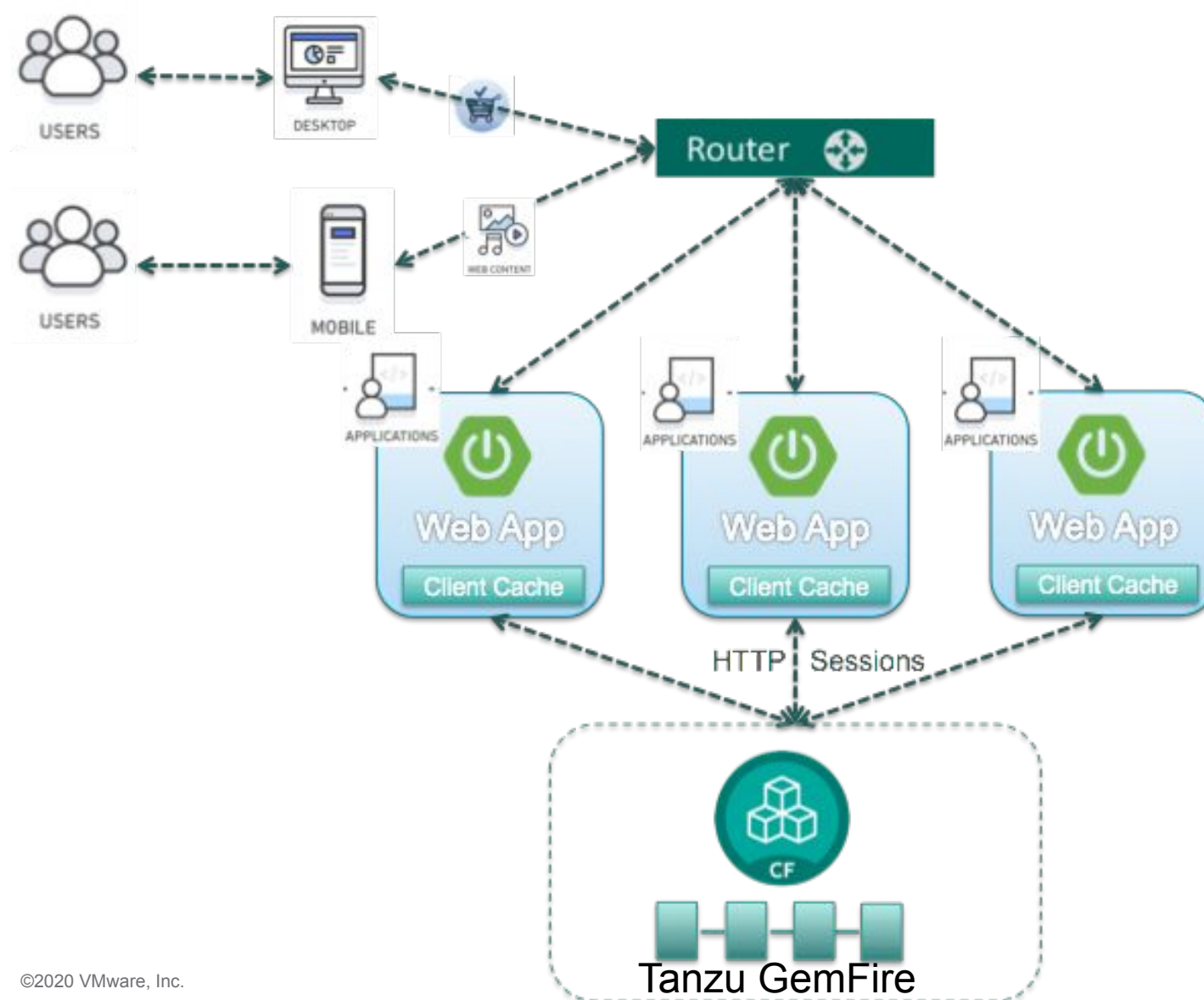
Tanzu GemFire Client Subscriptions



Tanzu GemFire Client Subscriptions



Design Pattern: Session State Caching



Distributed Session Caching



Scale-out Web Apps need
High Performance Data

- Web Session Replication
- Data Caching
- Pivotal Cloud Cache for Cloud Foundry

Lab 3



Look Aside Caching

Summary

Spring Boot Data Gemfire

Developer

How to use Spring Data and Spring Boot abstractions for Tanzu GemFire

How to build, run, and test your applications at scale

How developers can accelerate test cycles by implementing applications using mock objects as supported by the Spring Test project

Deployment

How availability of data can be maintained in light of node/pod failures and/or whole site failures with gemfire cluster of locators and servers.

How developers can use the Tanzu GemFire Operator to quickly instantiate GemFire

Scale

How to take advantage of Tanzu GemFire availability to build resilience into your applications on kubernetes.

How developers can configure Spring Caching applications to enable Tanzu GemFire for caching based on design patterns widely used in cloud native applications



Thank You