Building Cloud Native Applications at Scale with VMware Tanzu GemFire

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.

What is GemFire?

When is it Needed?

Why GemFire?

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

VMware Tanzu GemFire

GemFire is the best way to store the state for your 12 factor apps on the VMware Platform

Your app remains stateless
- but gives the appearance of a rich, stateful app. State is stored in a highly resilient, distributed in-memory cache

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

Cache your application data
- App data is available at memory speeds anywhere the app runs

Distributed in Memory Key Value Store Optimized for Apps Running on the VMware Platform

GemFire uses On Demand Service Broker for customized plans to support:

- 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

Pattern: Static or Slowly Changing Data

- 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.

Examples are 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.

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.

We know the number one reason for caching is performance

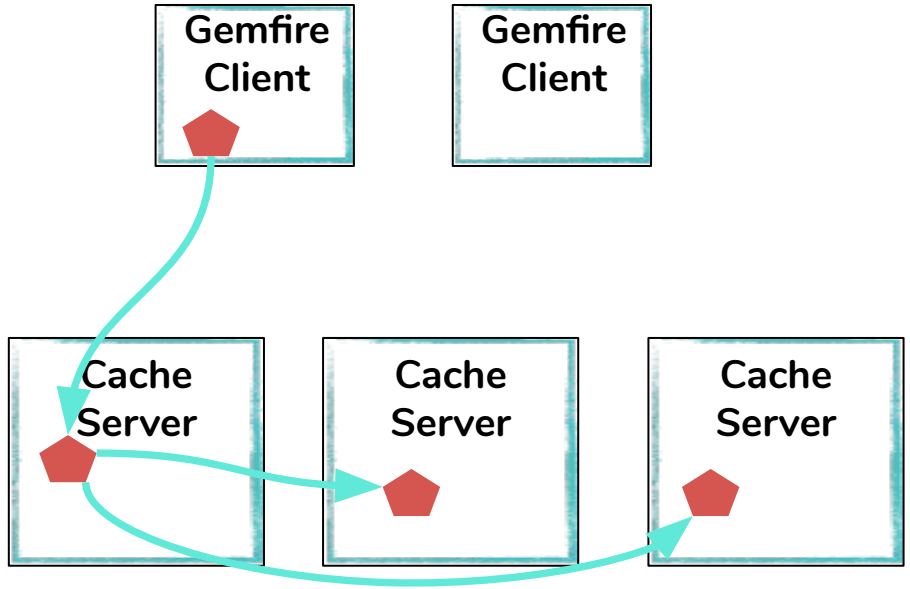
We are continually improving on our already impressive numbers.
For example in the most recent release we achieved…

- >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)

Consistency is a MUST

Our strong consistency is based on the synchronous replication in Geode

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



You ALWAYS get the CORRECT Data

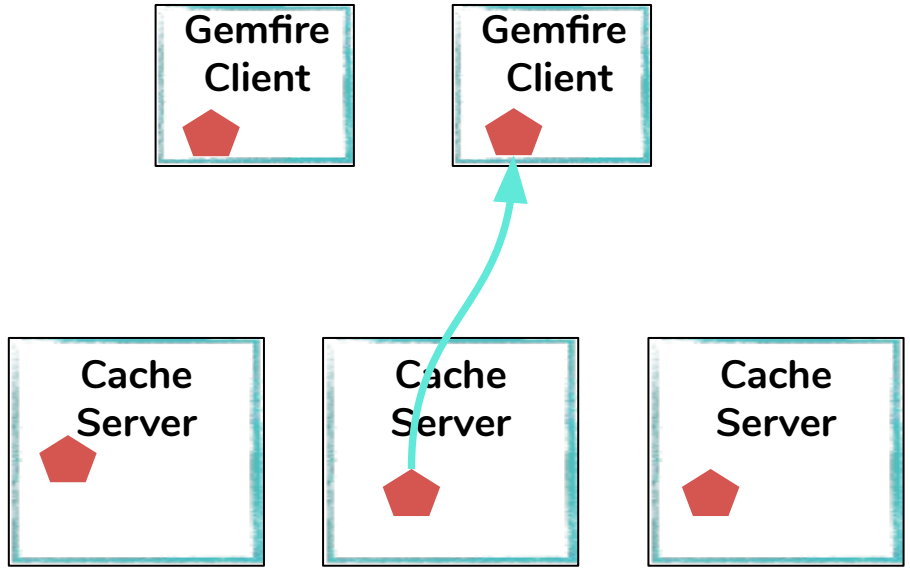
https://content.VMware.io/blog/acid-cap-and-pcc

Consistency is a MUST

Our strong consistency is based on the synchronous replication in Geode

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

https://content.VMware.io/blog/acid-cap-and-pcc

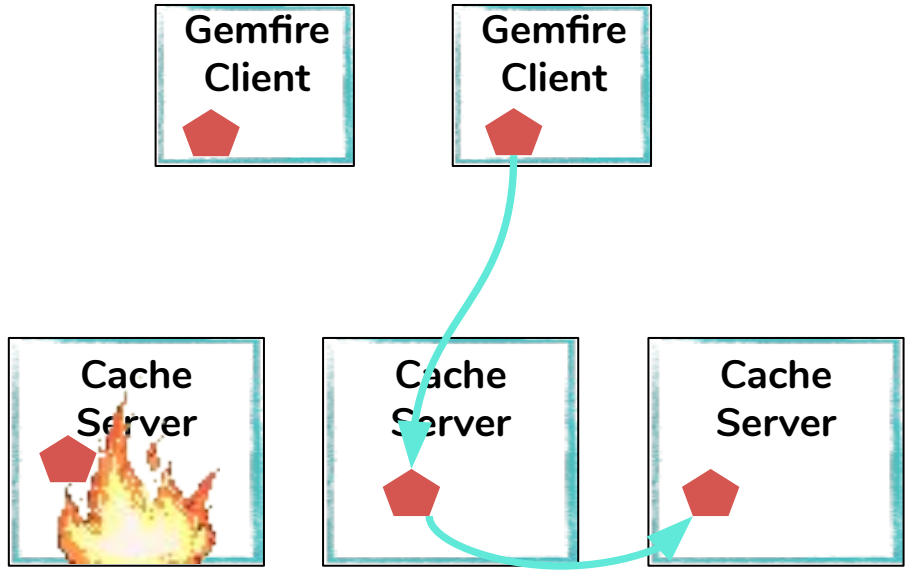
Consistency is a MUST

Our strong consistency is based on the synchronous replication in Geode

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 Primary fails new Primary that is
elected
already has the updated data



You ALWAYS get the CORRECT Data

https://content.VMware.io/blog/acid-cap-and-pcc

Consistency is a MUST

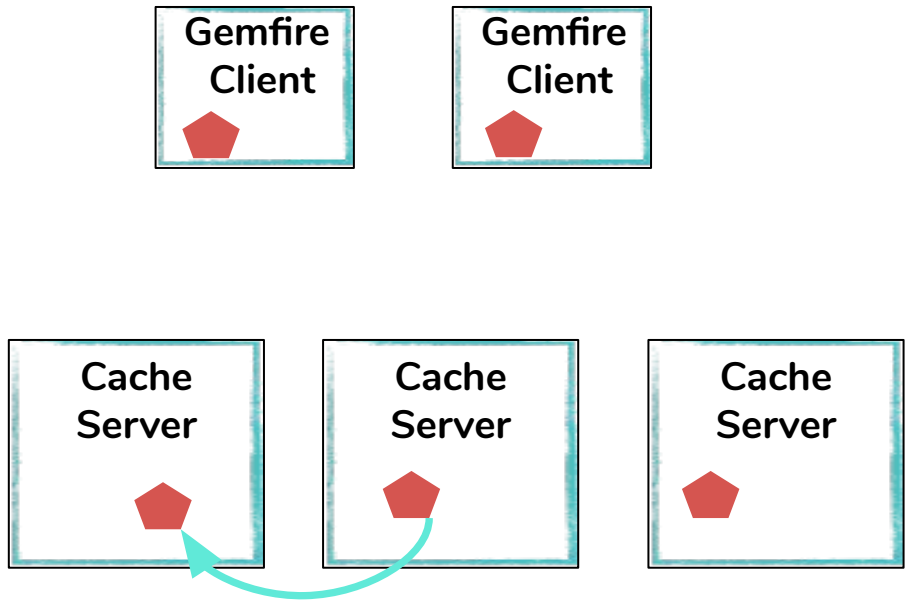
Our strong consistency is based on the synchronous replication in Geode

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 Primary fails new Primary that is
elected
already has the updated data

When server returns it is re-populated
with the latest updates from its peers



You ALWAYS get the CORRECT Data

https://content.VMware.io/blog/acid-cap-and-pcc

Value GemFire Brings

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

What is Geode and GemFire?

“Every modern application needs a Cache”

Apache Geode and GemFire are the same codebase.

GemFire expands on the Apache Geode codebase by providing enterprise ready features including:

- ➔ OOTB Security implementation
- ➔ Day 2 operations automation
- ➔ Easy logging and metrics
- ➔ Easy Upgrades
- ➔ Easy installations

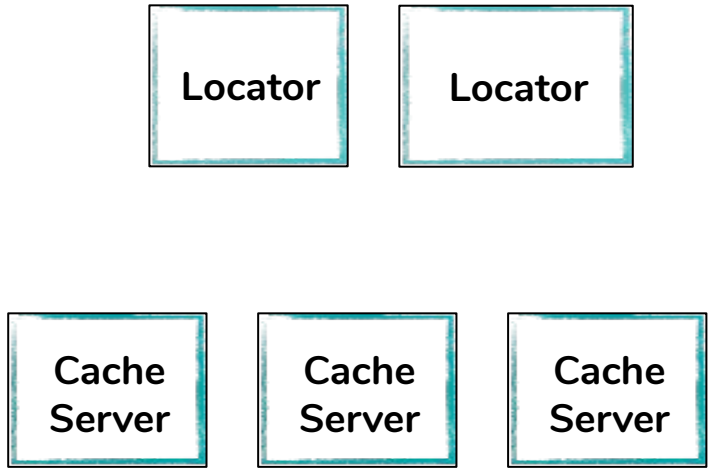
Enabling developers to create on demand instances automatically.

GemFire Topology

GemFire Members

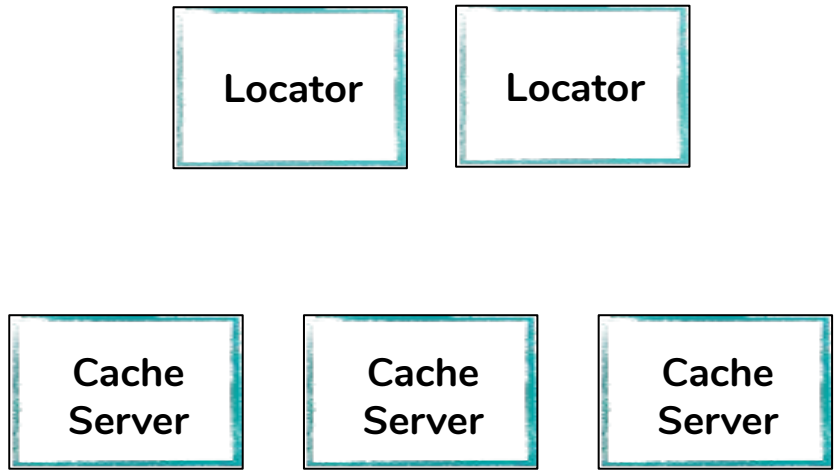
Locator

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



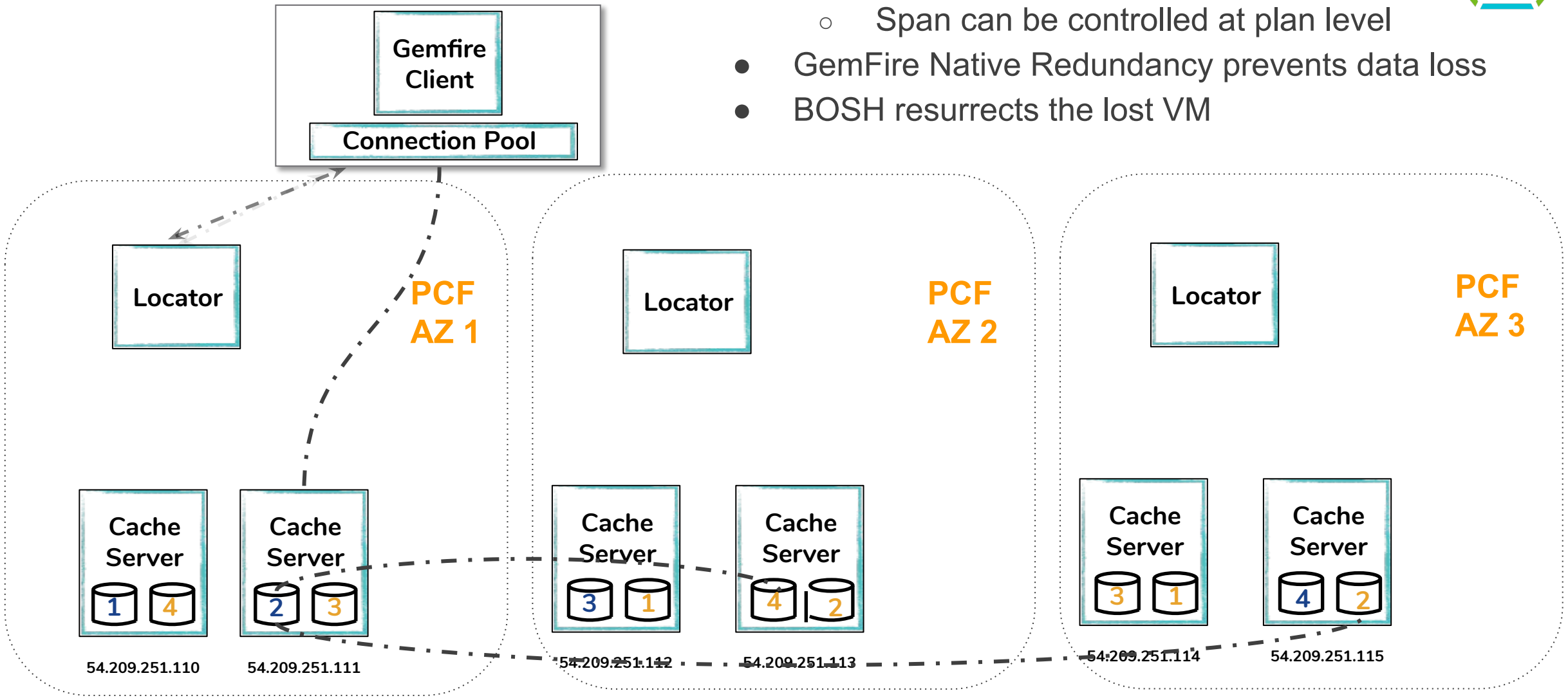
Cache Server

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



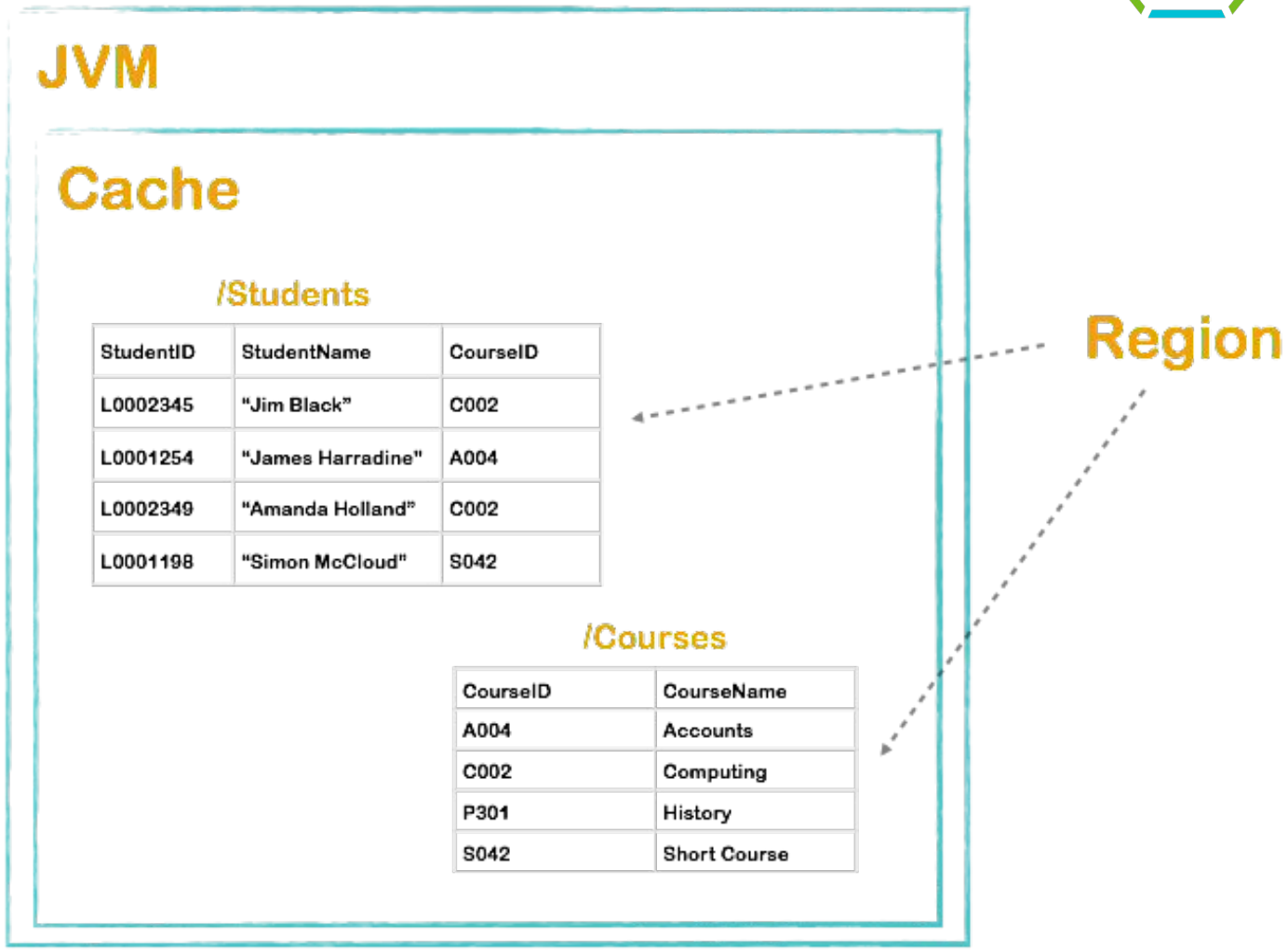
GemFire Topology

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



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

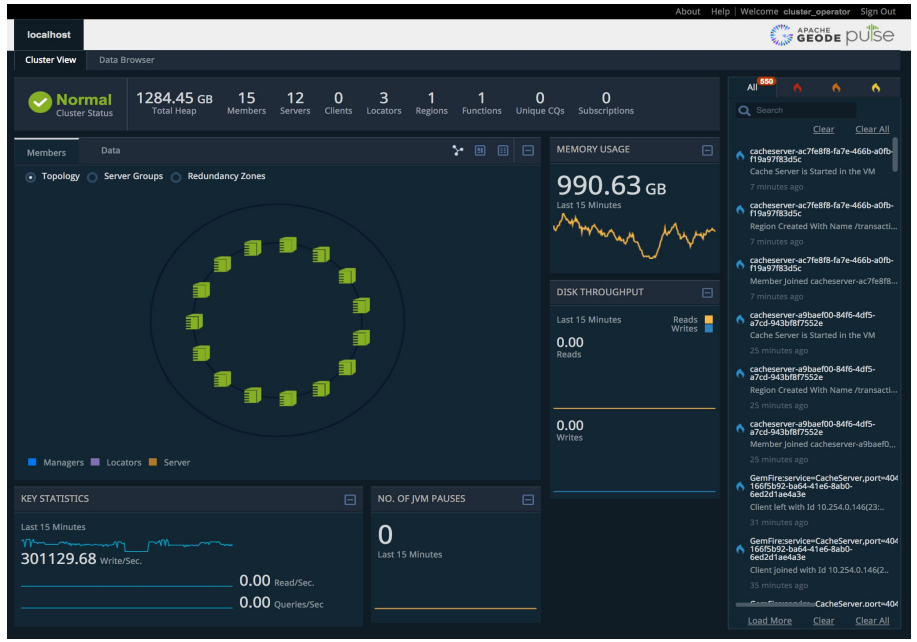


Gemfire Cluster Management

Gemfire Shell (GFSH)

- Cluster Administration
- Service Control
- Full Operation Support

PULSE



- Cluster Health Monitor
- Region Query

VMware Tanzu GemFire for Kubernetes

GemFire brings an in-memory key-value store to Kubernetes. Tanzu GemFire is powered by Apache Geode

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.

Workshop Github Repo

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

Lab 1

Spring Boot Data Gemfire Essentials

Application Read/Write into 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 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 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 Gemfire


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

public class CustomerServiceApplication {
}
```

SBDG Annotation for 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 Annotation for 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 Gemfire Repositories


```
public interface CustomerRepository
extends CrudRepository<Customer,
Long> {

Customer findByNameLike(String
name);

}
```

Inherits:

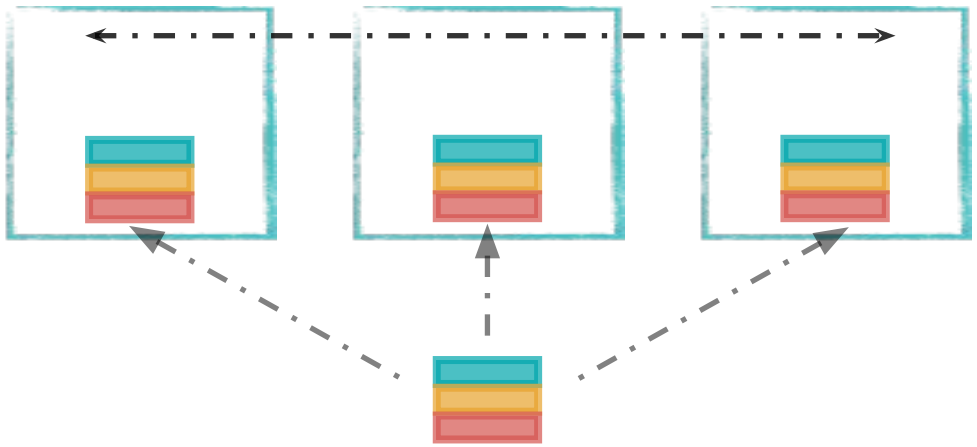
- **count()**
- **delete(T)**
- **deleteAll()**
- **deleteAll(Iterable<? extends T>)**
- **deleteById(ID)**
- **existsById(ID)**
- **findAll()**
- **findAllById(Iterable<ID>)**
- **findById(ID)**
- **save(S)**
- **saveAll(Iterable<S>)**

Types of Gemfire Regions

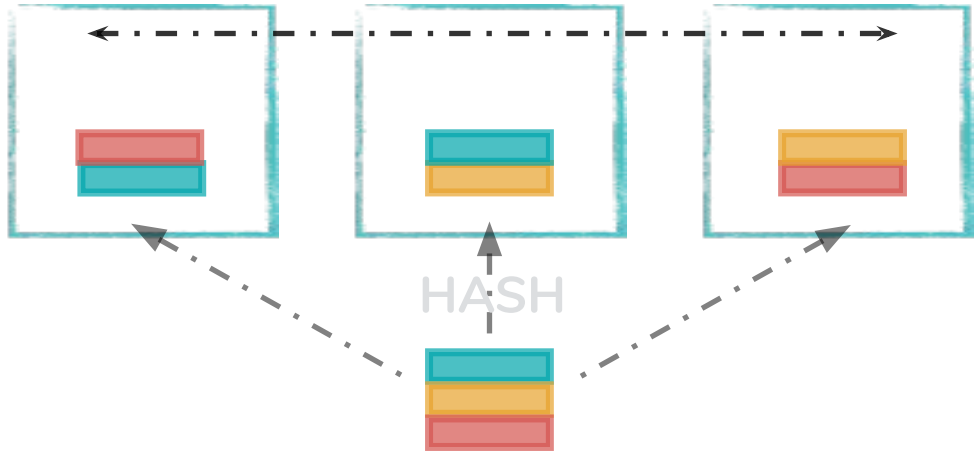
- Partitioned
- Replicated
- Disk Persisted
- WAN Replicated

Gemfire Regions

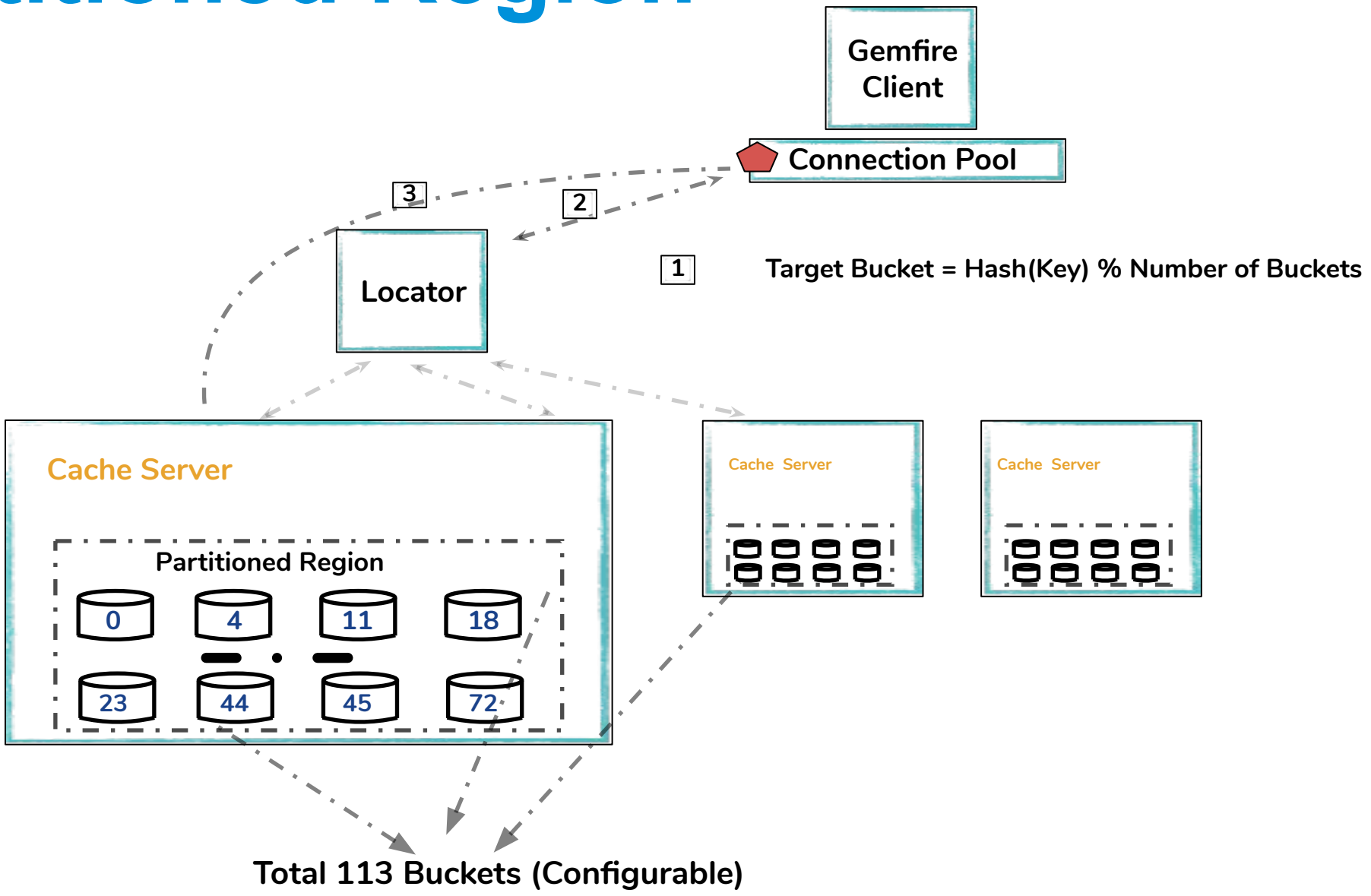
Replicated



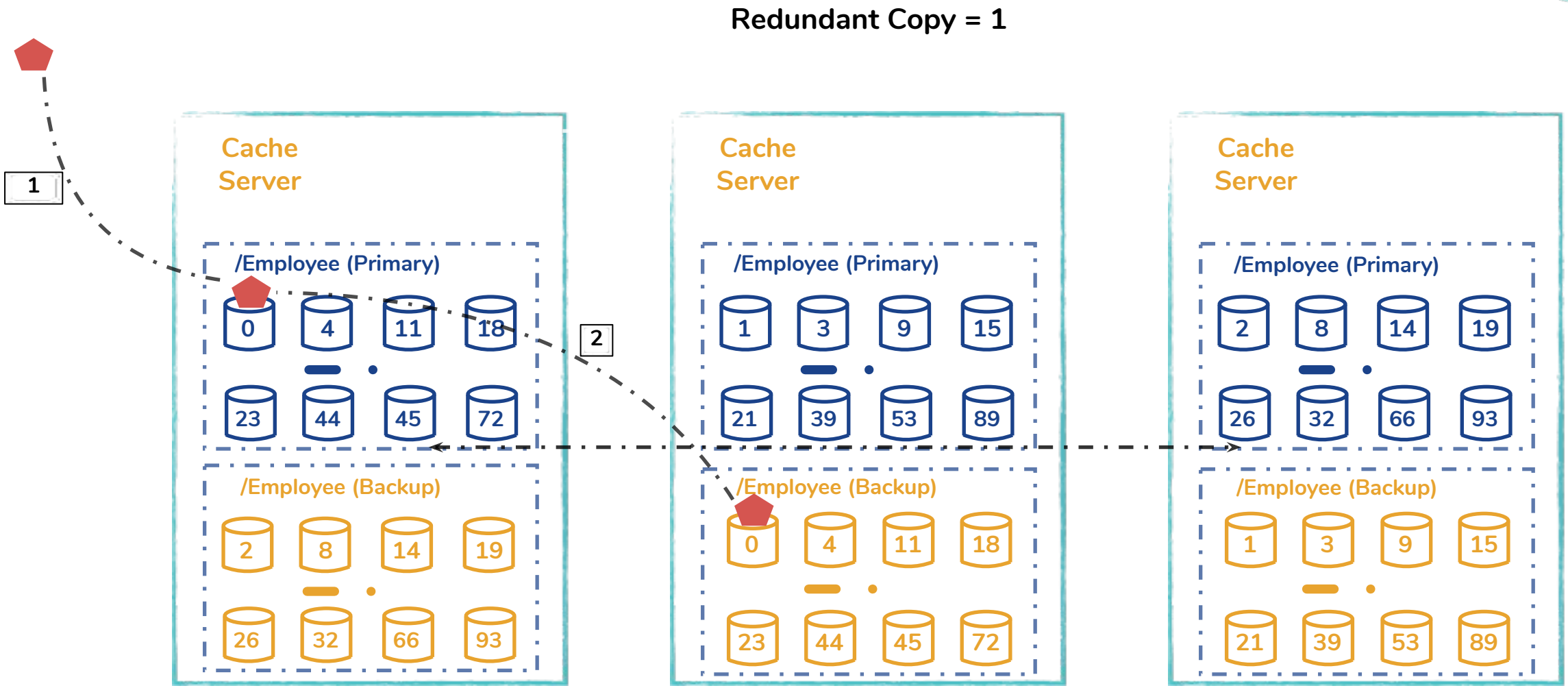
Partitioned



Partitioned Region

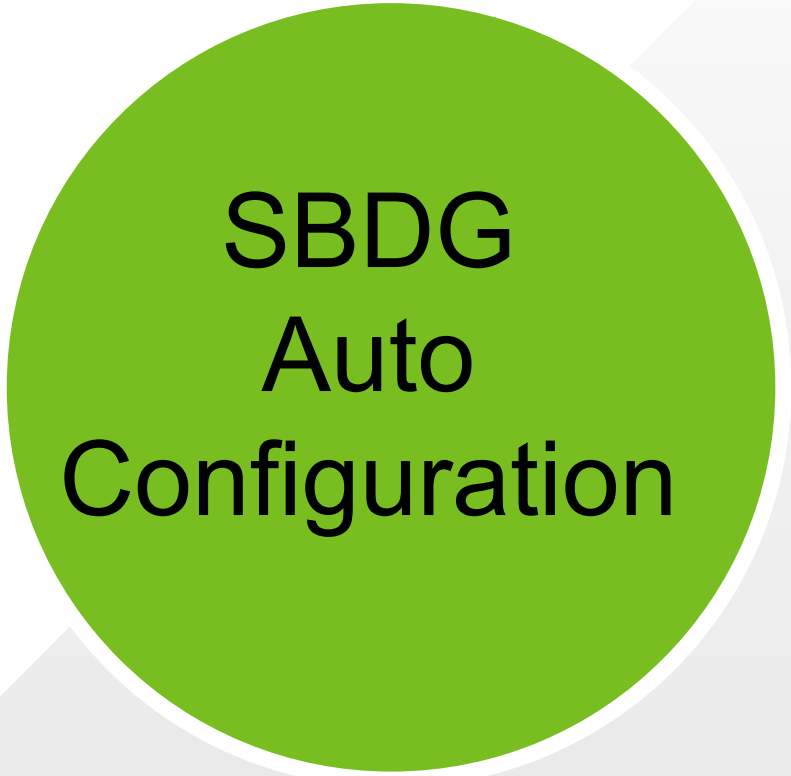


Partitioned Region



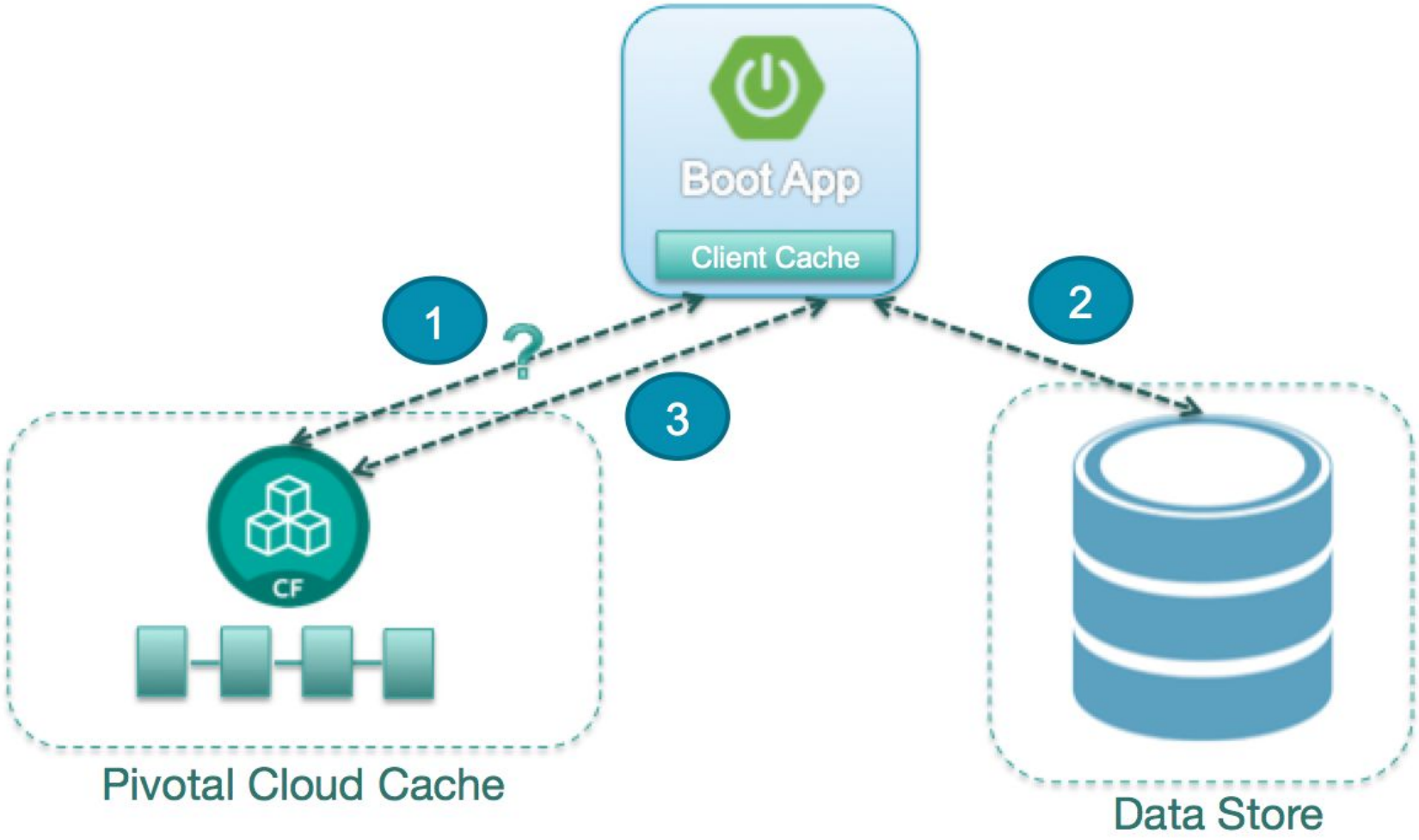
Client gets an ACK only when all the redundant copies are created

Lab 2



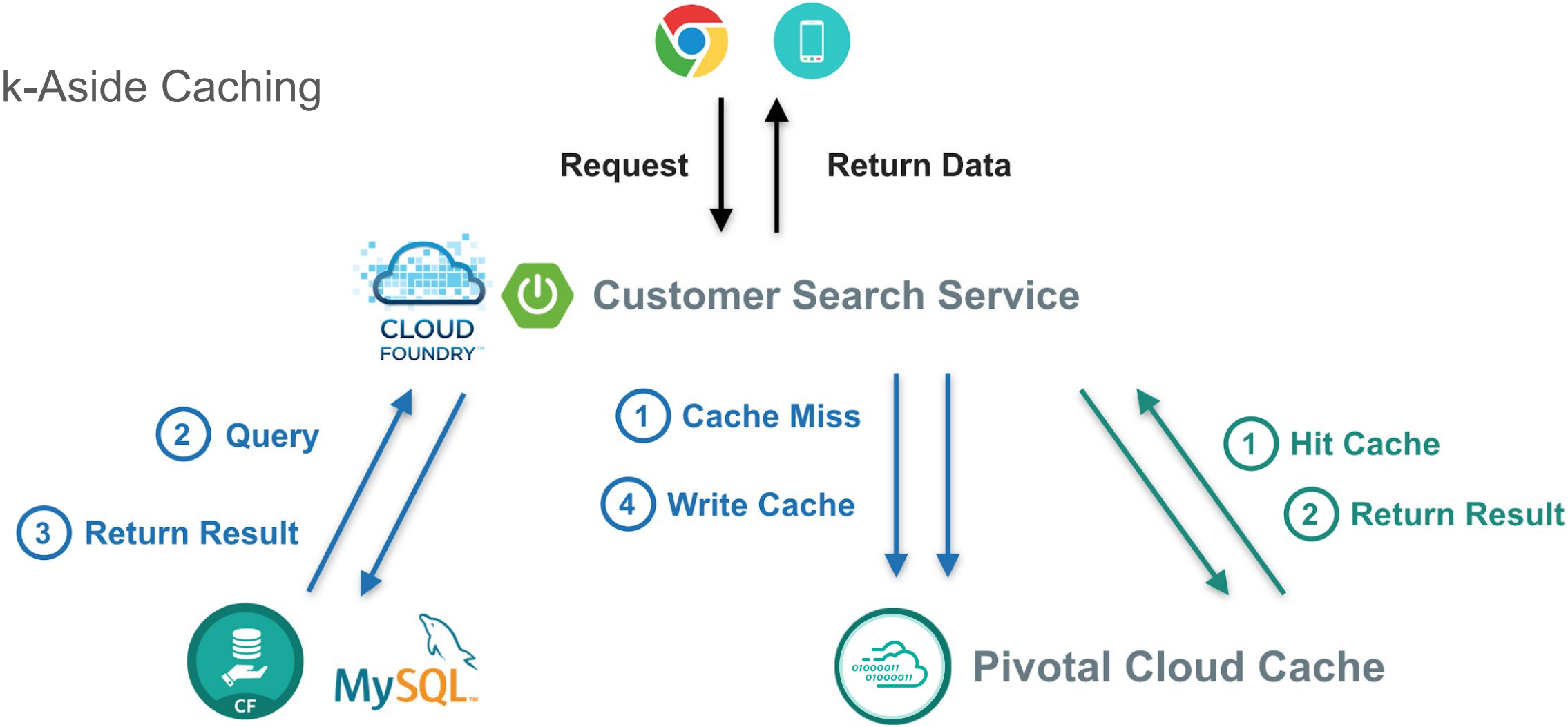
Gemfire Design Patterns

Design Pattern: Look-Aside Caching



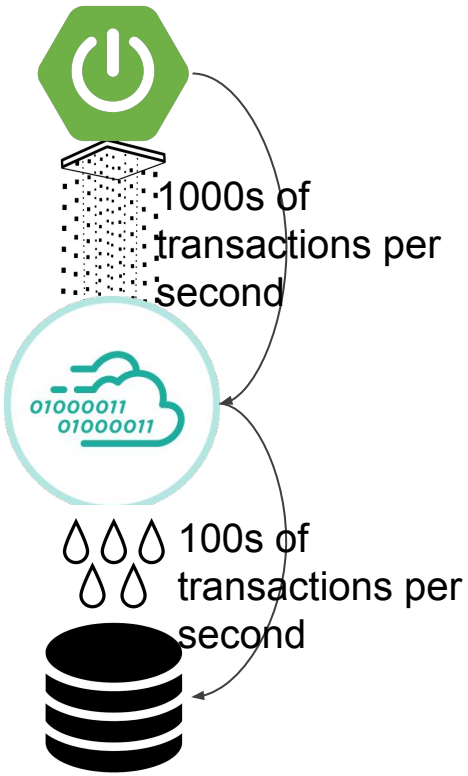
Demo App: Customer Search

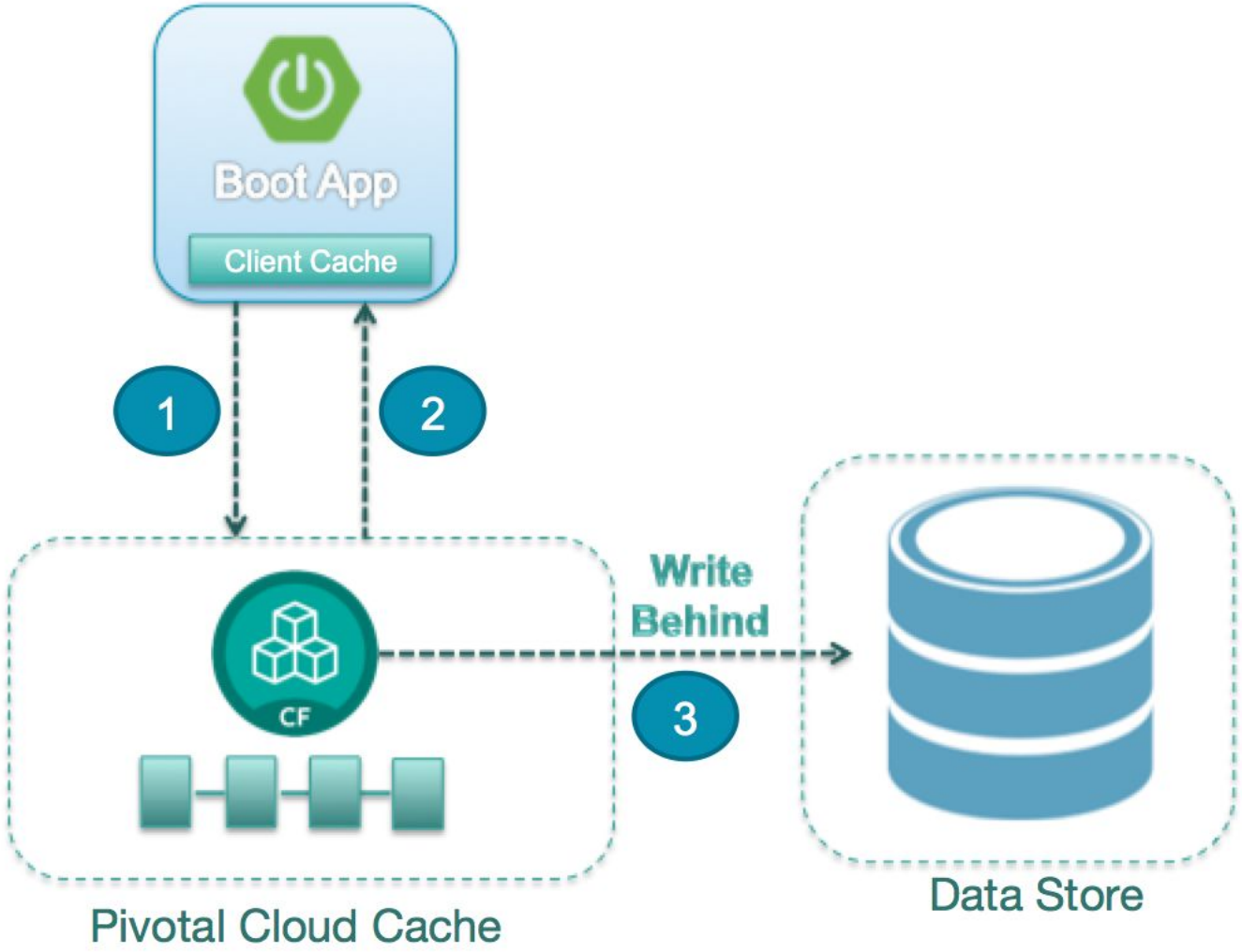
Look-Aside Caching



Design Pattern: In-line Caching

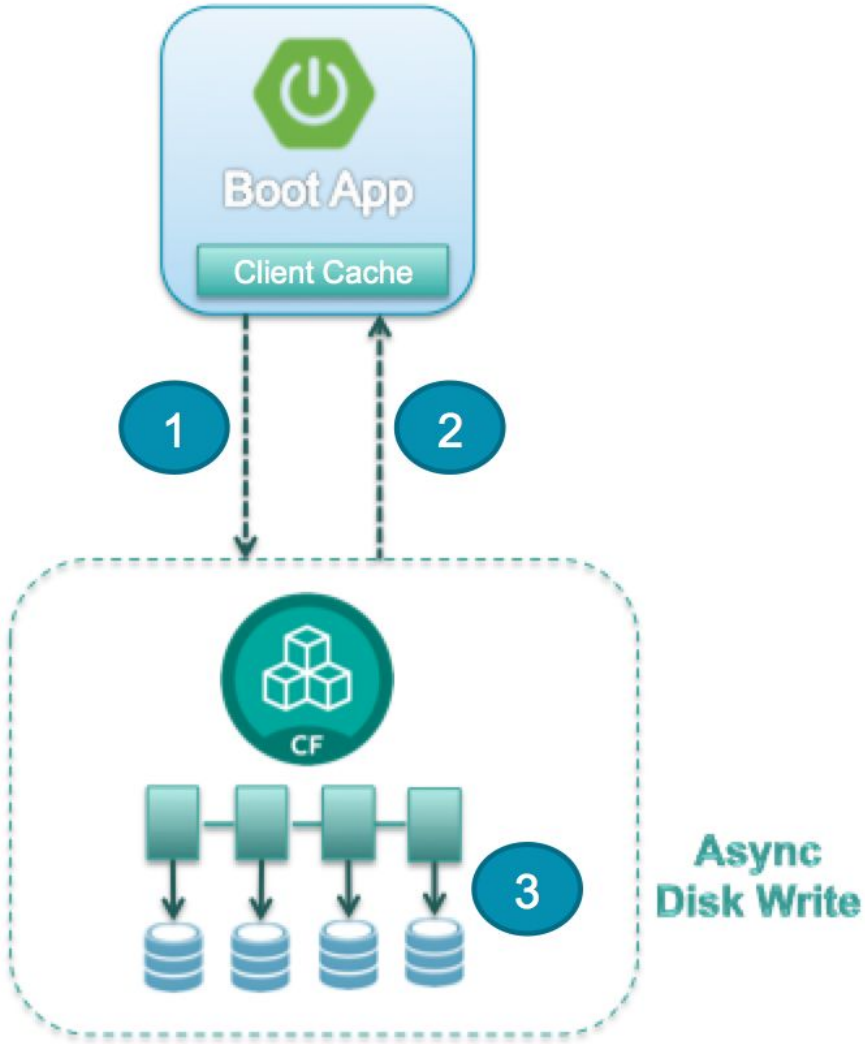
Fast KV Store front-ending RDBMS





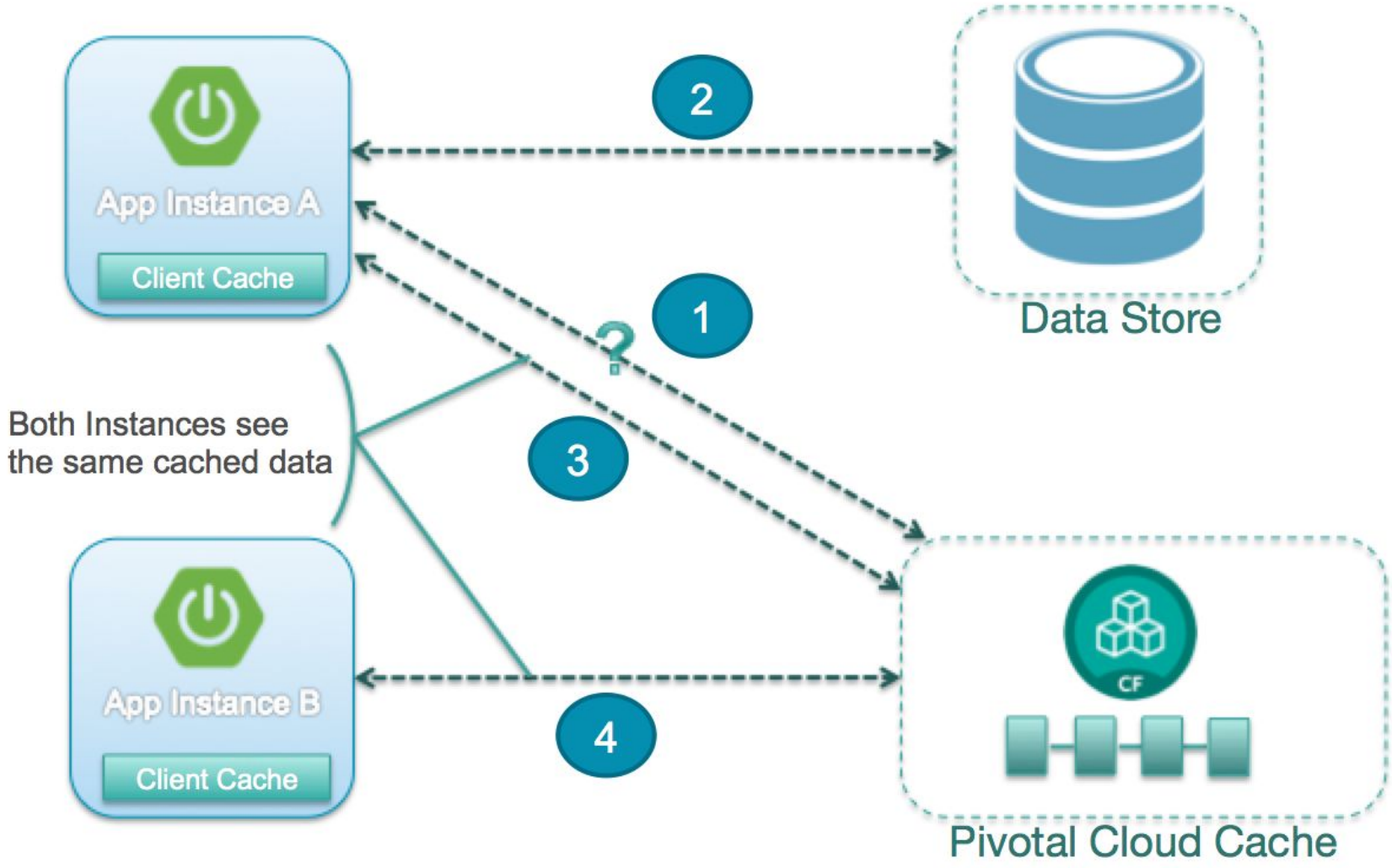
Design Pattern: Disk Persistence

High Availability with I



Pivotal Cloud Cache

Design Pattern: Shared Cache

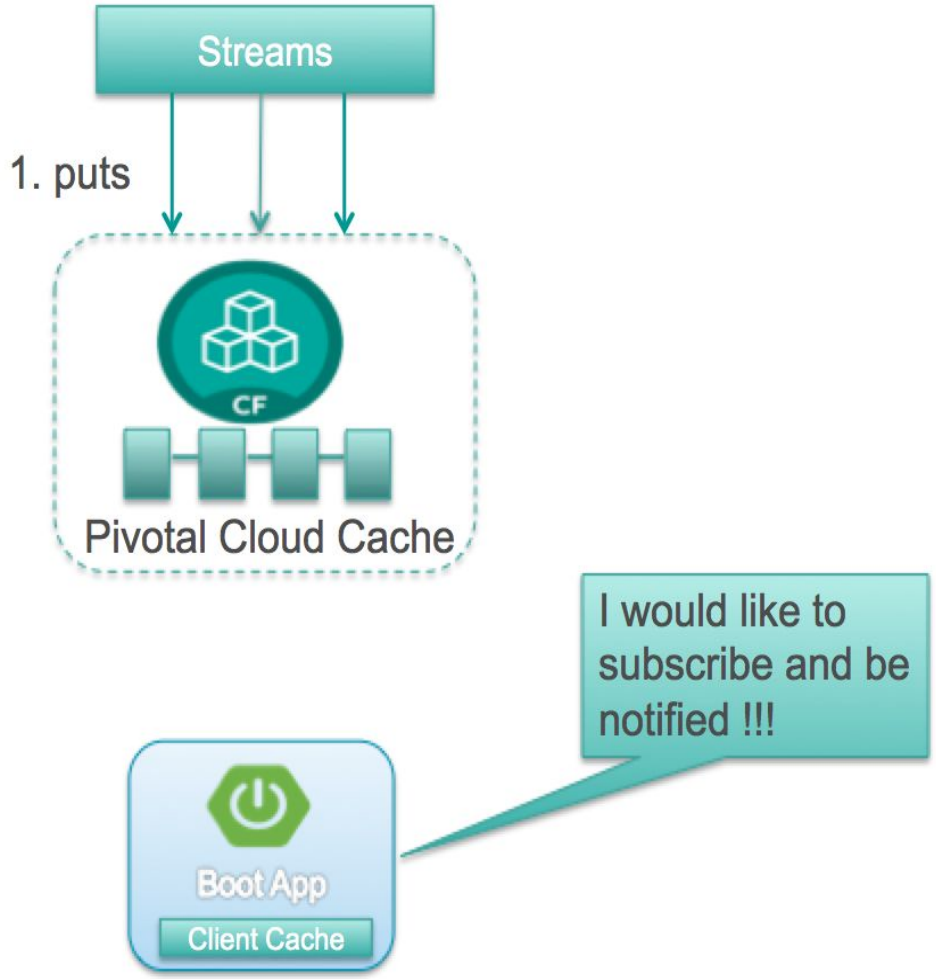


Design Pattern: Client Side Events

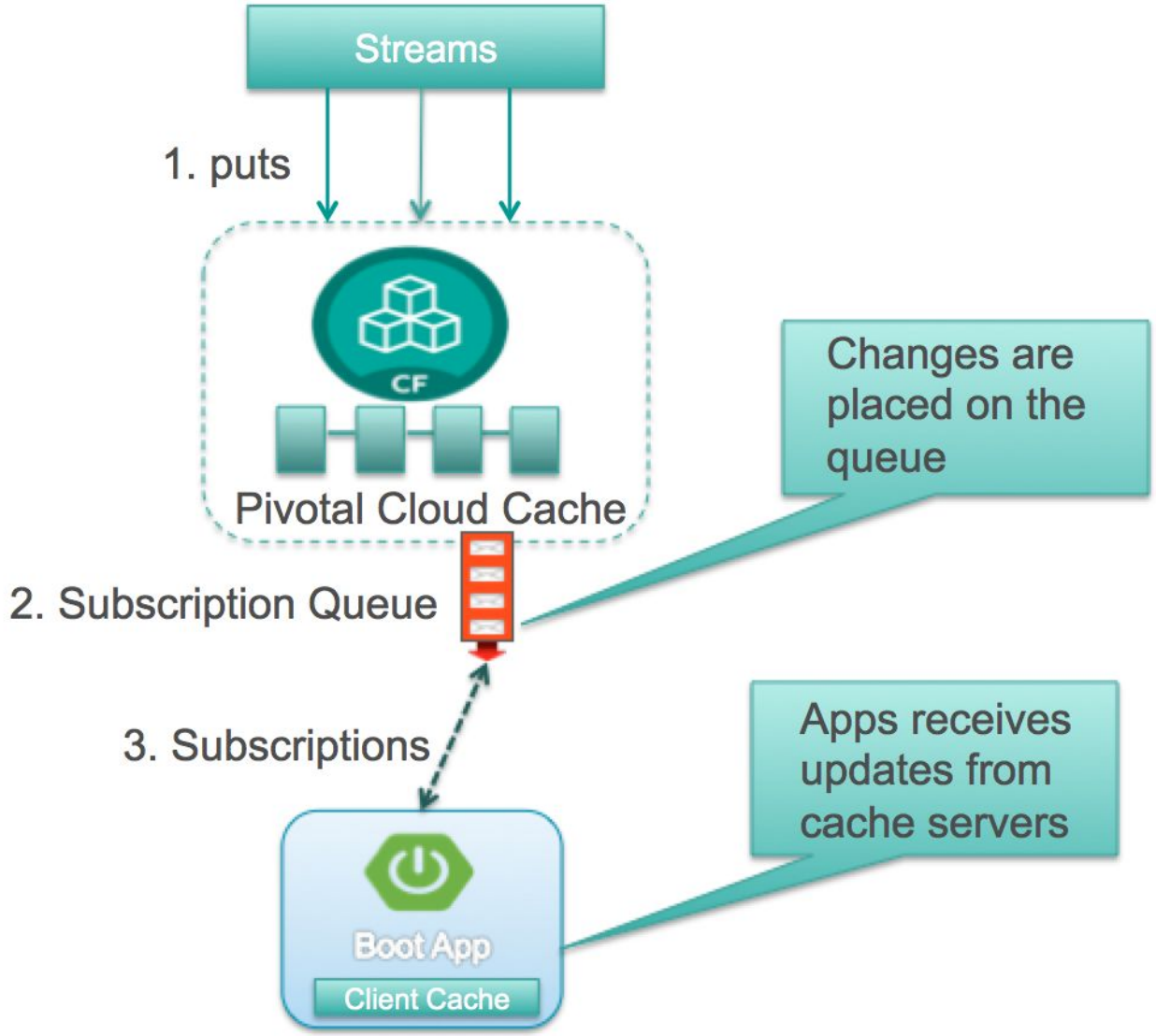
- **Client Subscriptions**
  - Client interest in server side event changes
  - Filter events based on RegEx Interest

- **Continuous Query(CQ)**
  - Register interest through server side queries
  - Suitable for creating dashboards

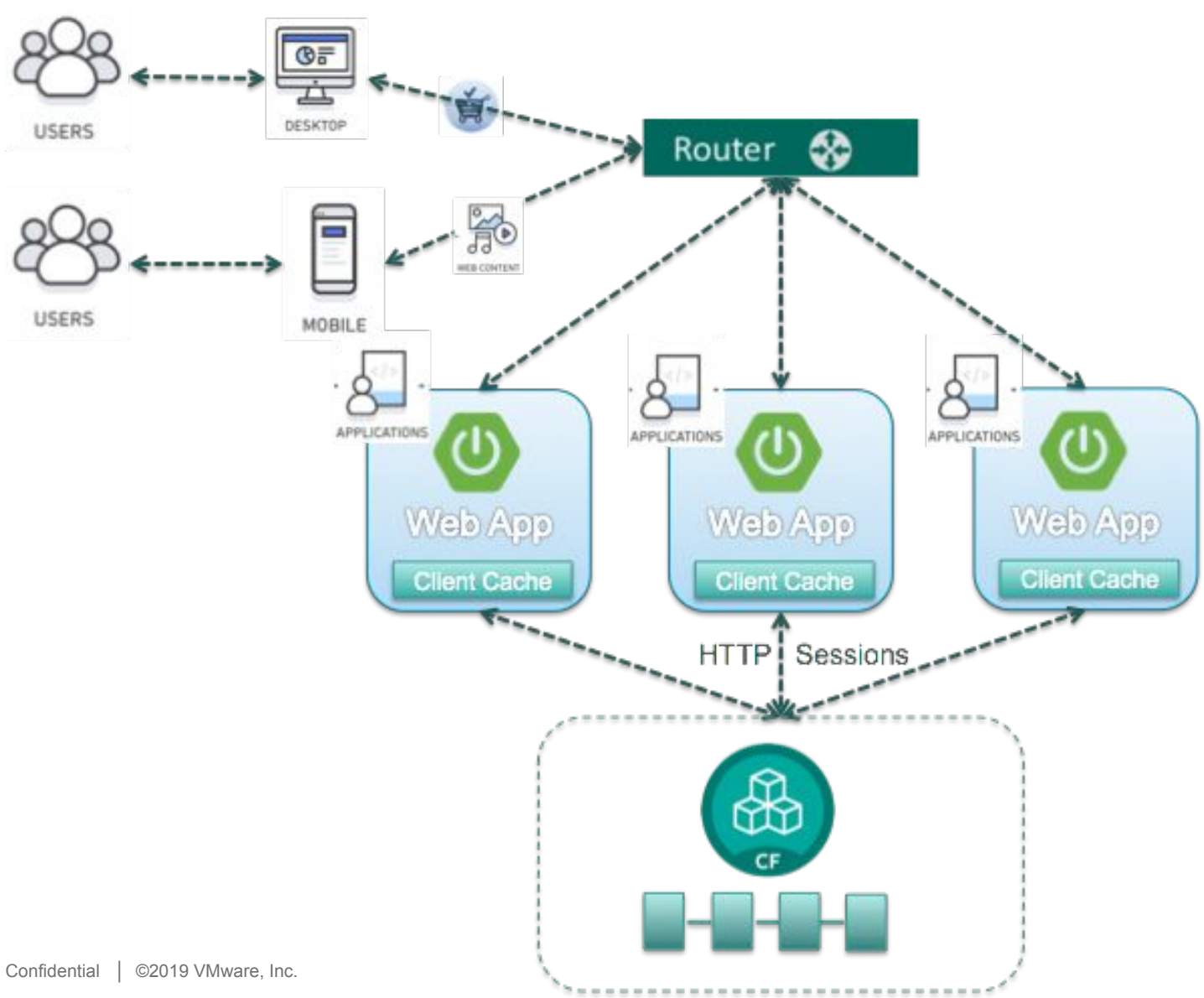
Gemfire Client Subscriptions



Gemfire Client Subscriptions



Design Pattern: Session State Caching

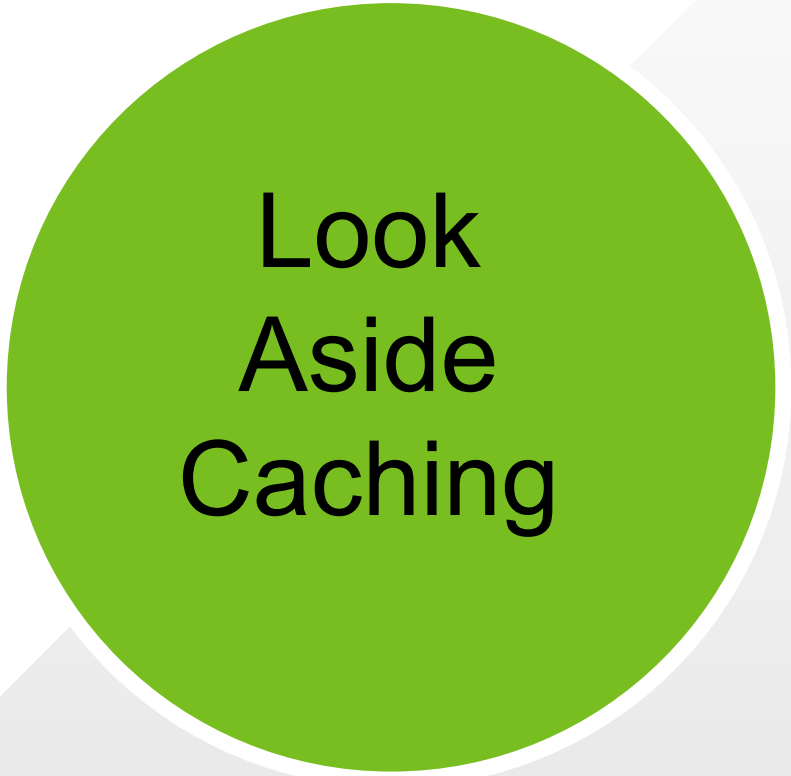


Distributed Session Caching

Scale-out Web Apps need High Performance Data

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

Lab 3



Thank You