

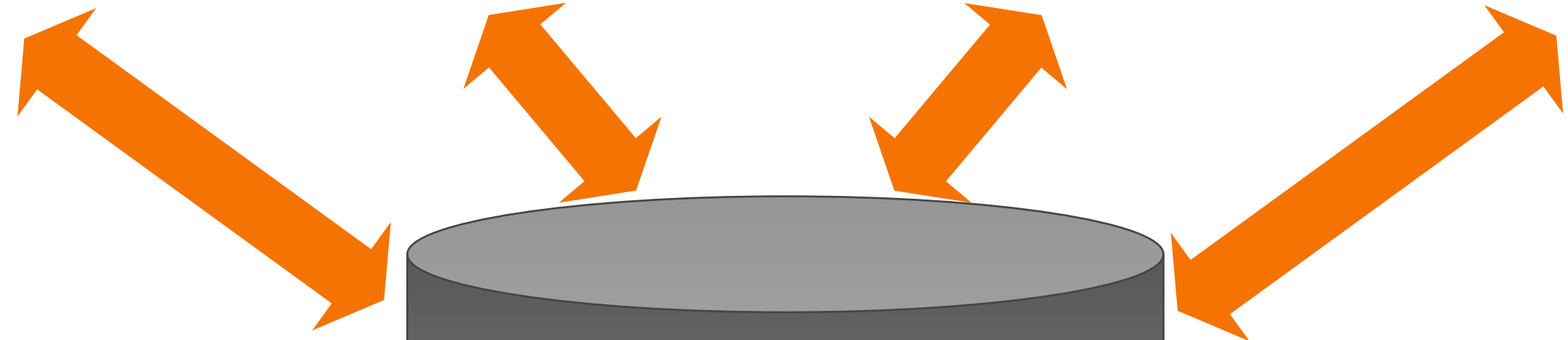
# The Essence of Caching

Greg Luck, Founder and CTO Ehcache, Terracotta

JavaOne 2011

Session 24241

# The Problem



>100 ms



Average Response Time

Speed



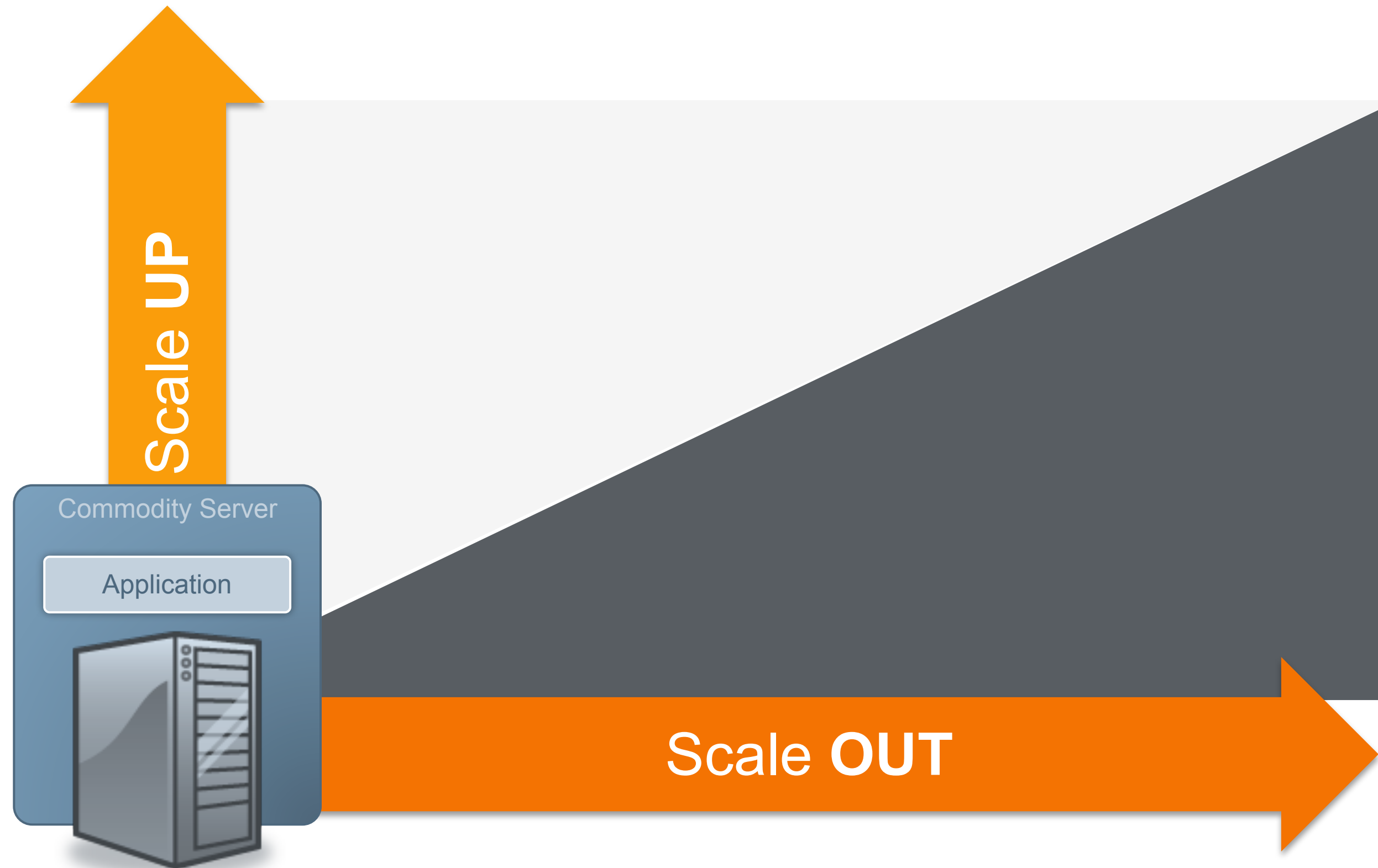
Cost



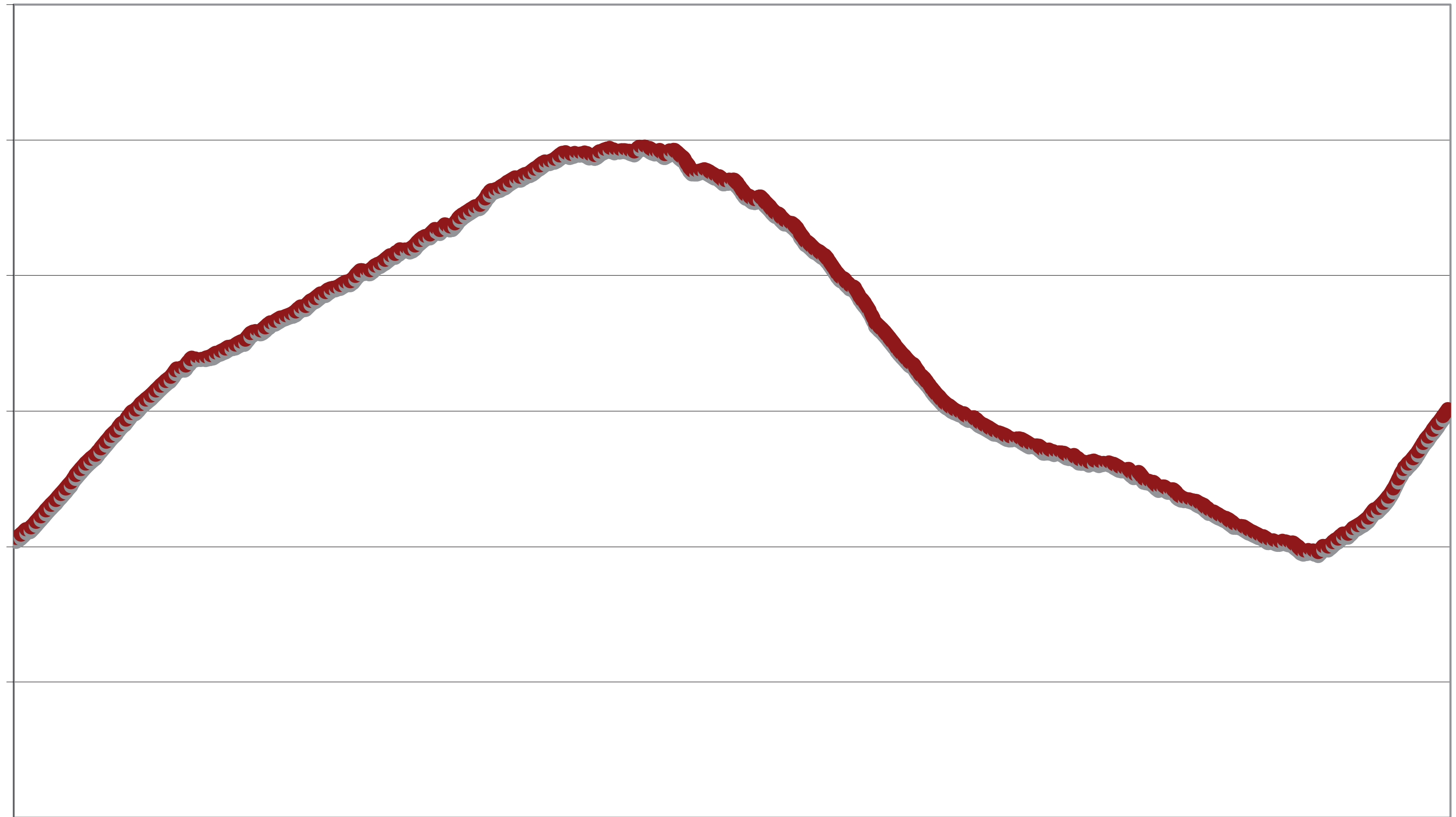
Scalability



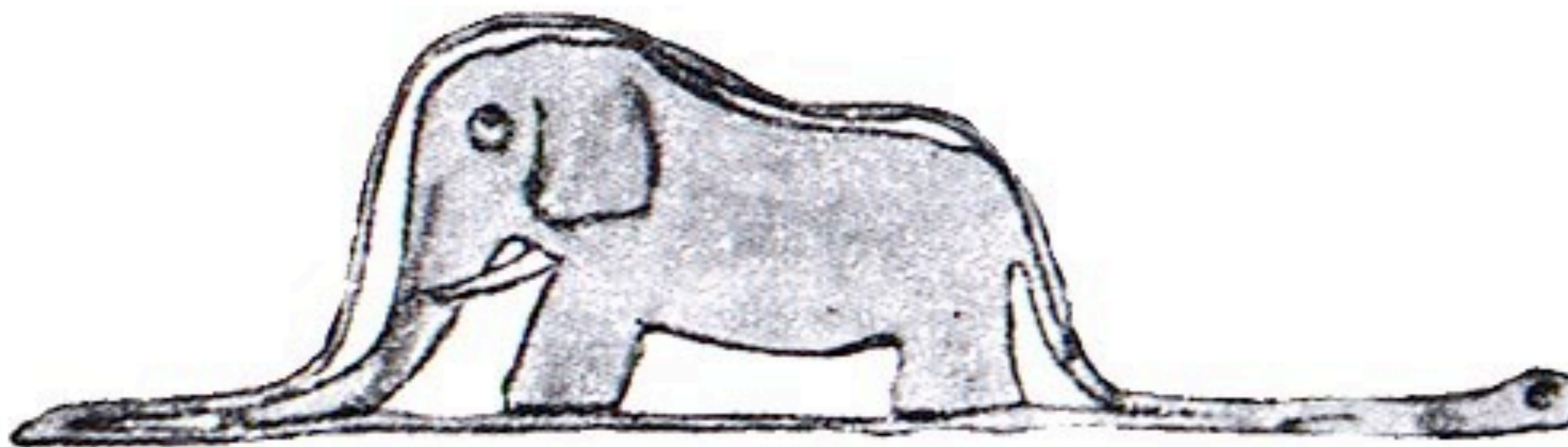
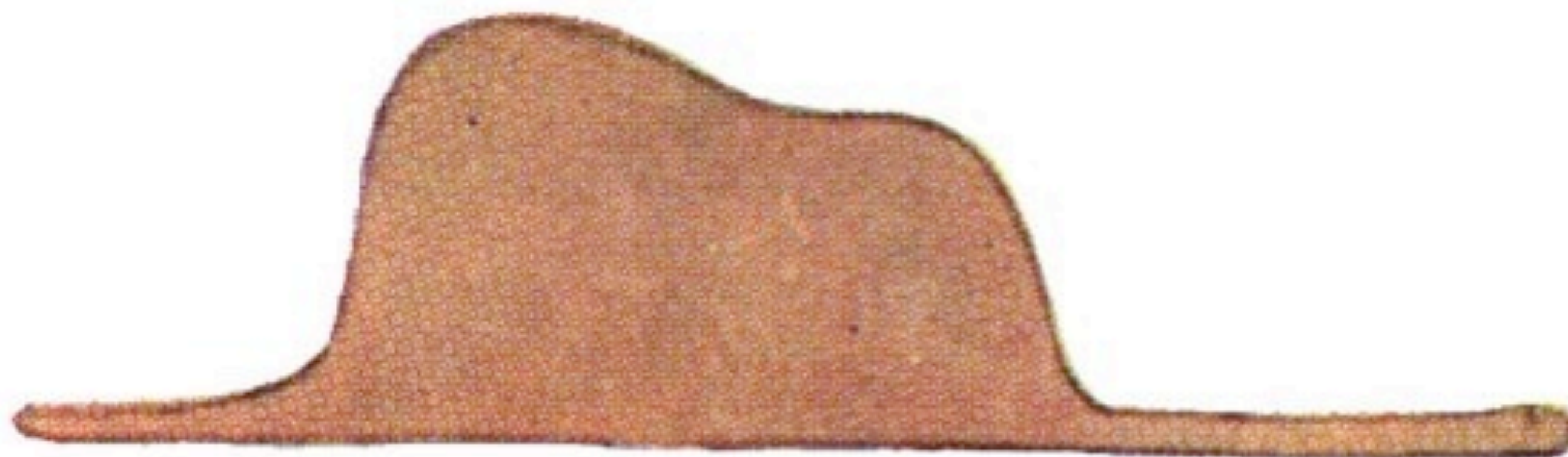
# Types of Scaling



# Capacity Planning - Peak Load



Google's Web Searches (1 Datacenter)



The Elephant Curve

# Desirable Properties of a Solution





# A Performance Problem Solving Methodology



# Amdahl's Law

$$\text{Speedup} = \frac{1}{(1 - \text{Proportion Sped Up}) + \frac{\text{Proportion Sped Up}}{\text{Speed up}})}$$

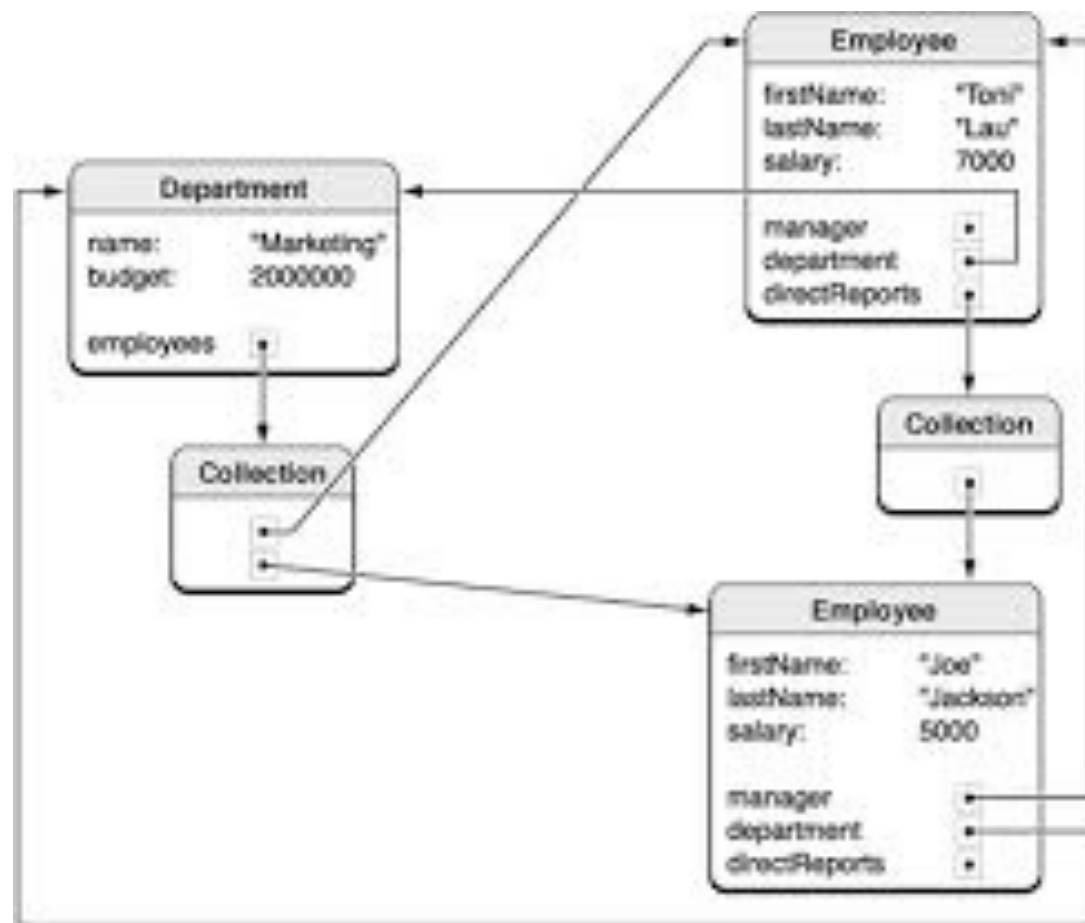
# Rules of Thumb



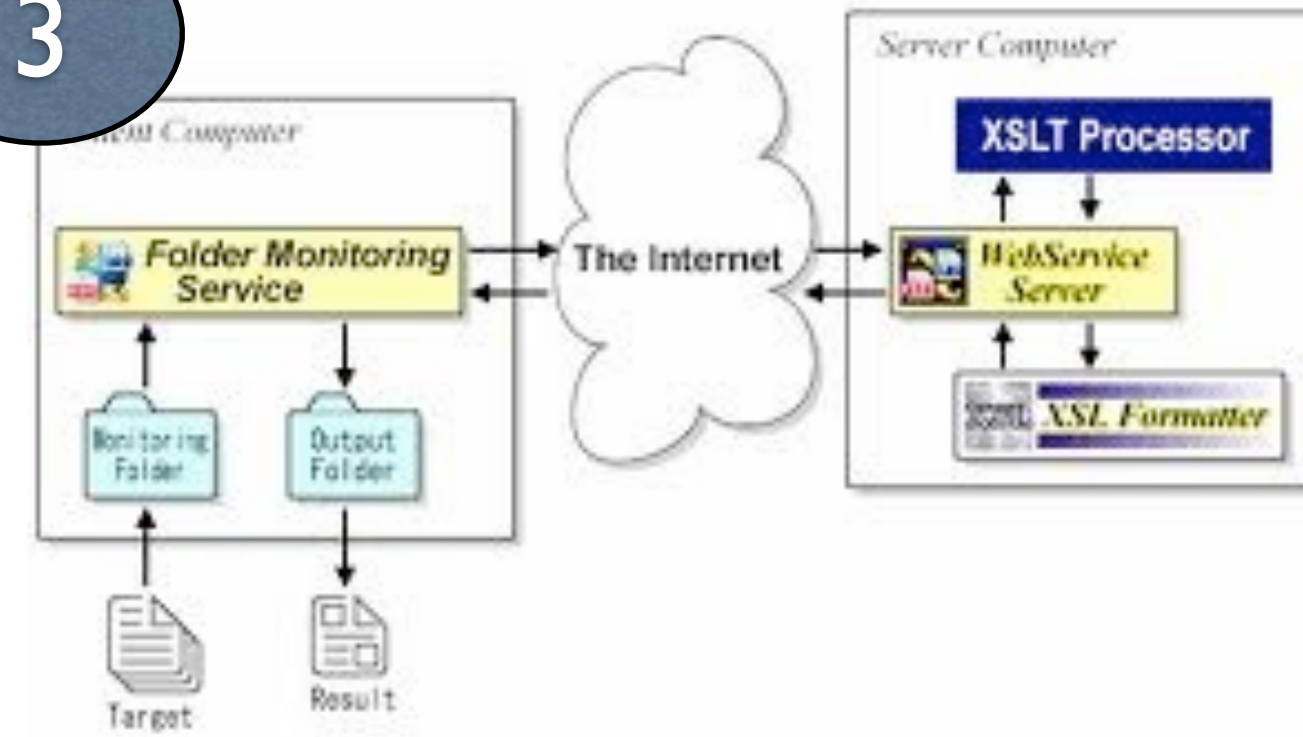
1



2



3



4

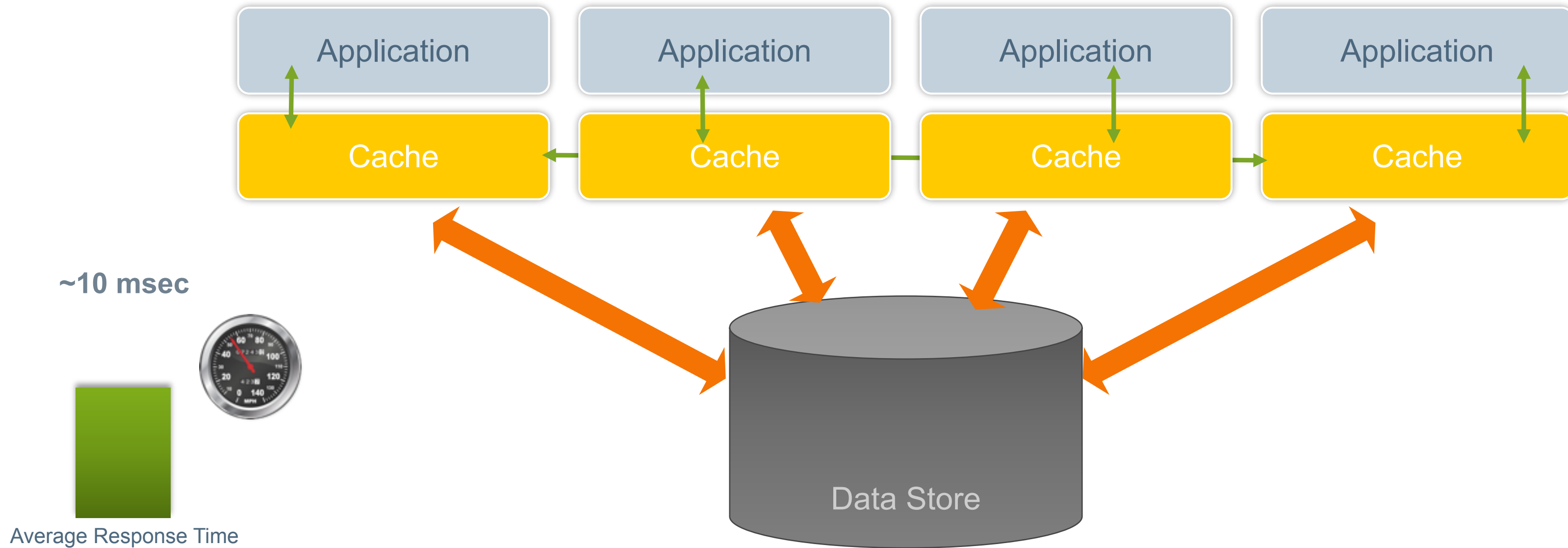


Enter Caching

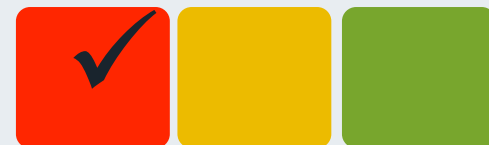
# Caching

- Fastest To Implement
- Offload
- Performance
- Scale up
- Scale out (Distributed Caches Only)
- Buffer against load variability

# Caching Layer is a temporary store



Speed



Costs



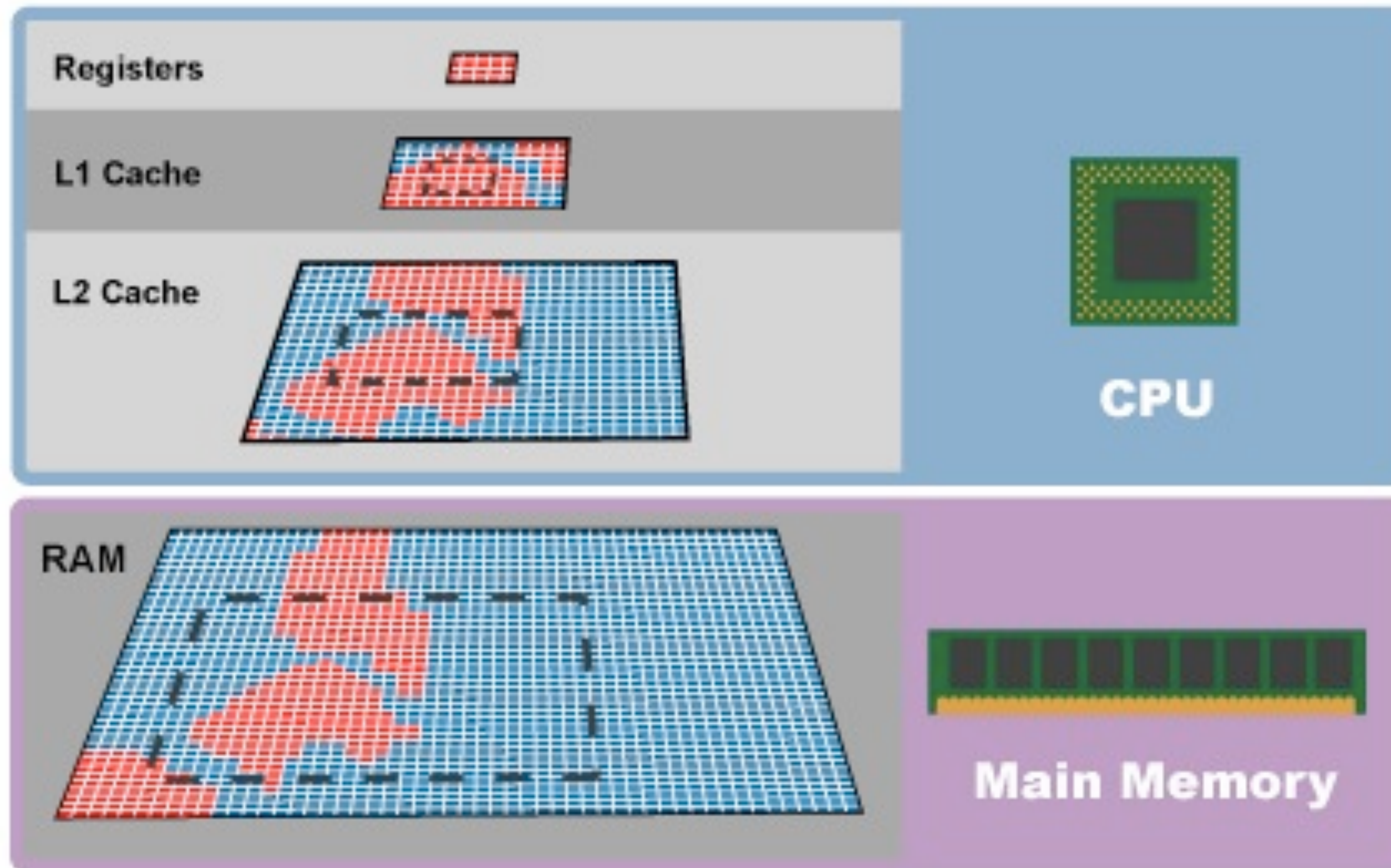
Scalability



Moving data from the database into the cache increases processing speed and can reduce database licensing and maintenance costs.

# Cache Design Forces

# Hardware Storage Design

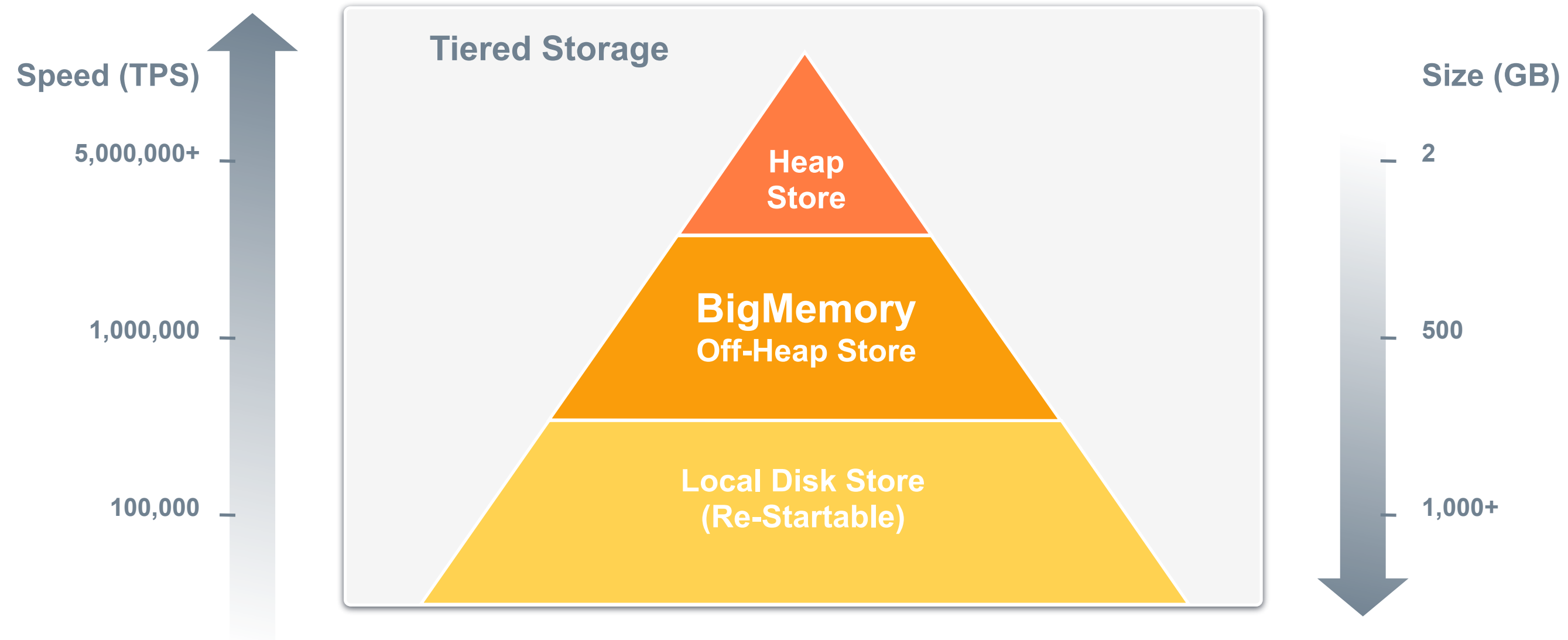


# Hardware Storage Design

Level	Access Time	Typical Size	Technology	Managed By
<b>Registers</b>	1-3 ns	?1 KB	Custom CMOS	Compiler
<b>Level 1 Cache</b> (on-chip)	2-8 ns	8 KB-128 KB	SRAM	Hardware
<b>Level 2 Cache</b> (off-chip)	5-12 ns	0.5 MB - 8 MB	SRAM	Hardware
<b>Main Memory</b>	10-60 ns	64 MB - 1 GB	DRAM	Operating System
<b>Hard Disk</b>	3,000,000 - 10,000,000 ns	20 - 100 GB	Magnetic	Operating System/User

Reference: <http://arstechnica.com/old/content/2002/07/caching.ars/2>

# Ehcache Local Storage Design



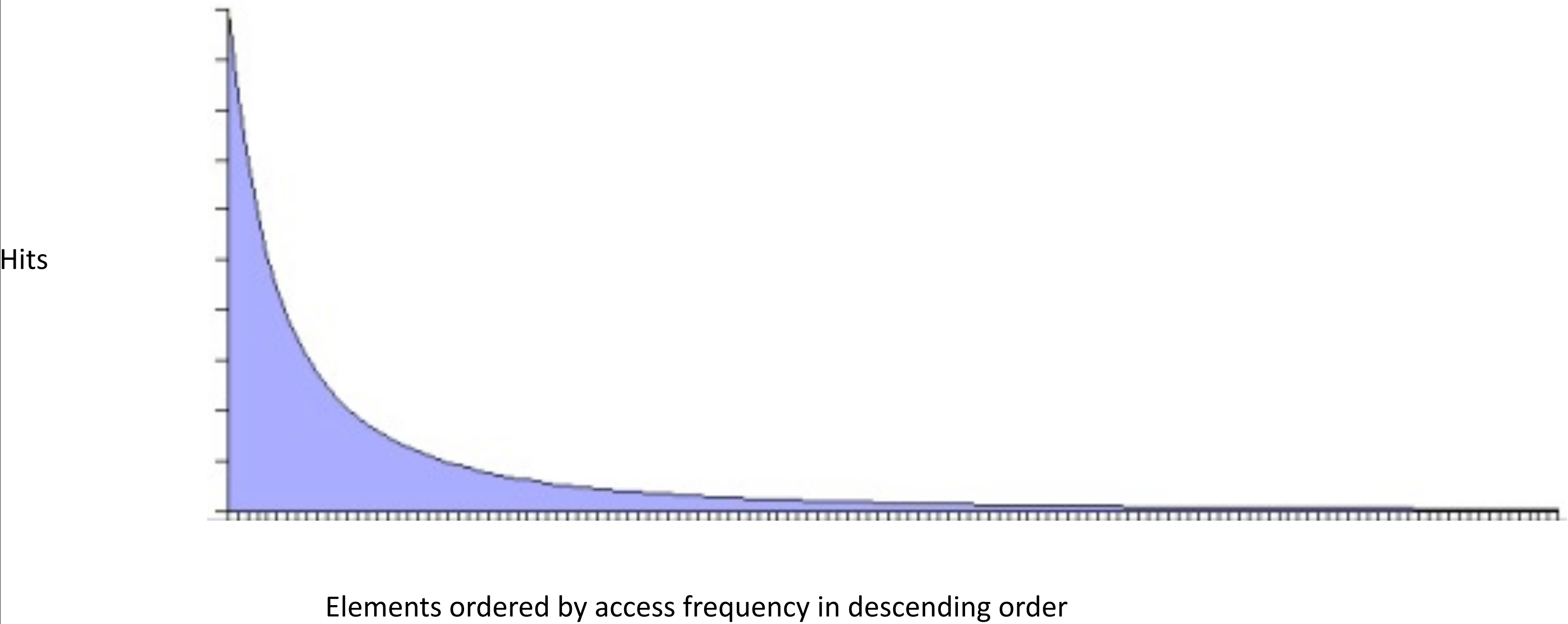
# Fitting It All In

# Not If all data was created equal

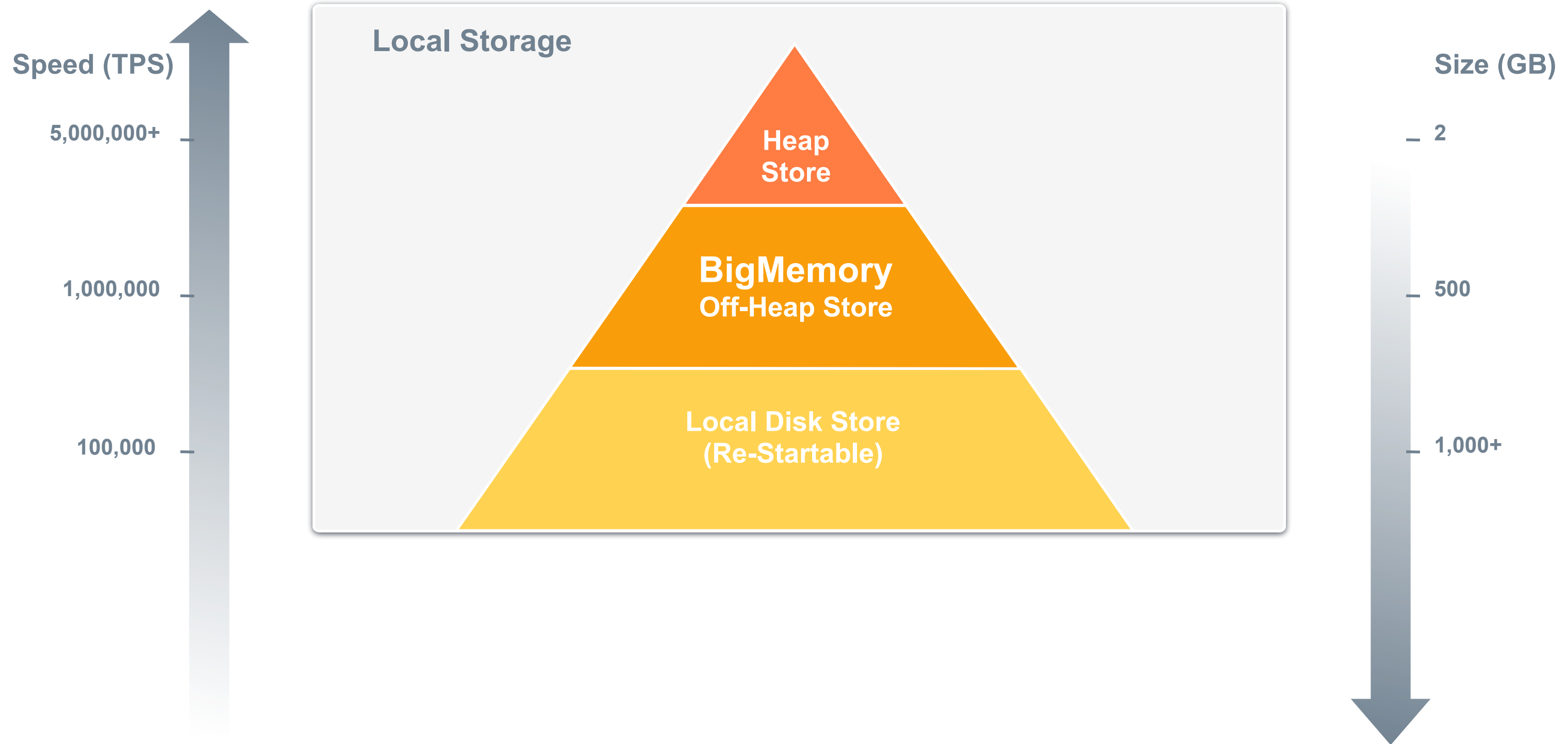


Elements ordered by access frequency in descending order

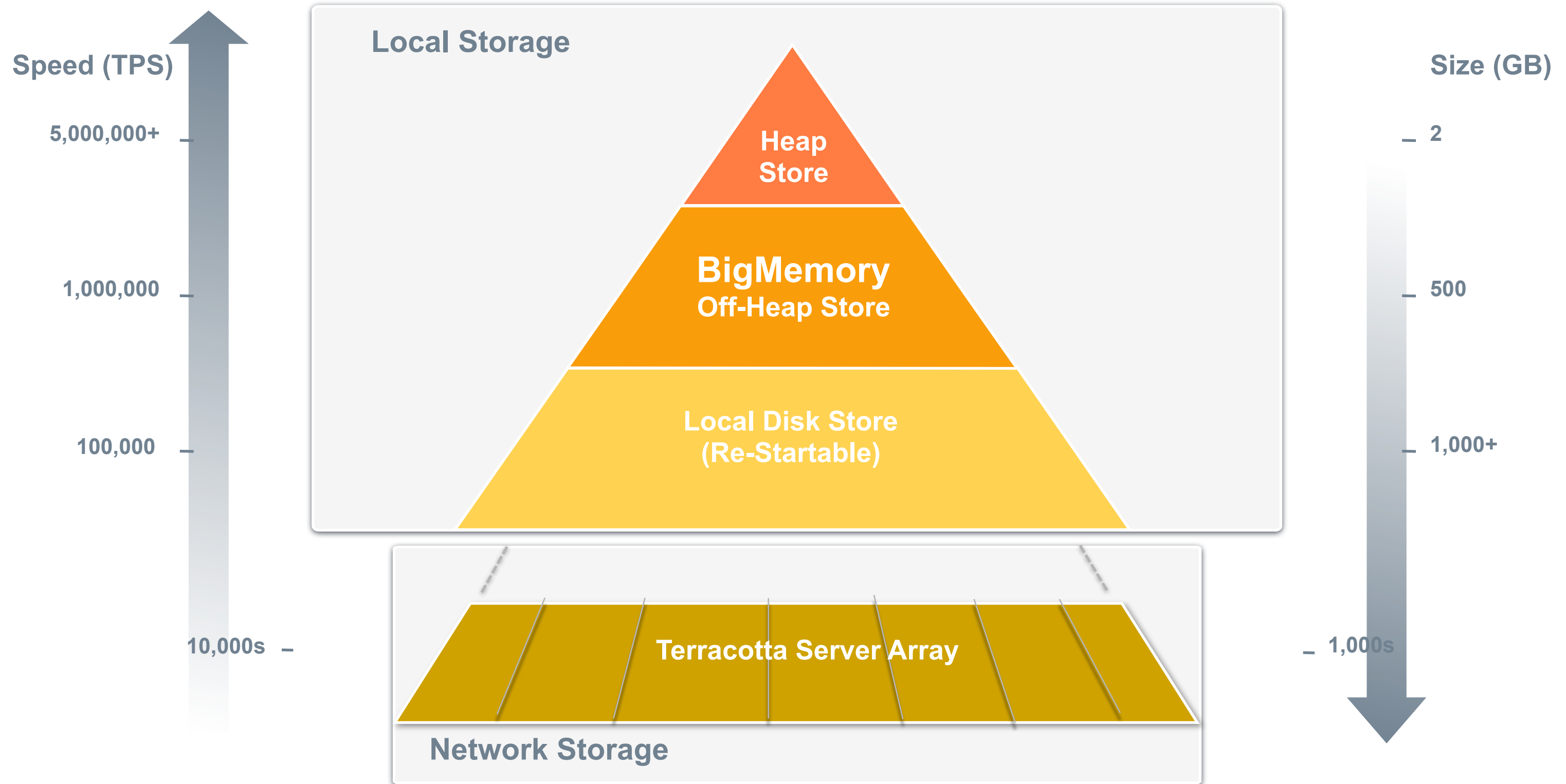
# Not all data is created equal



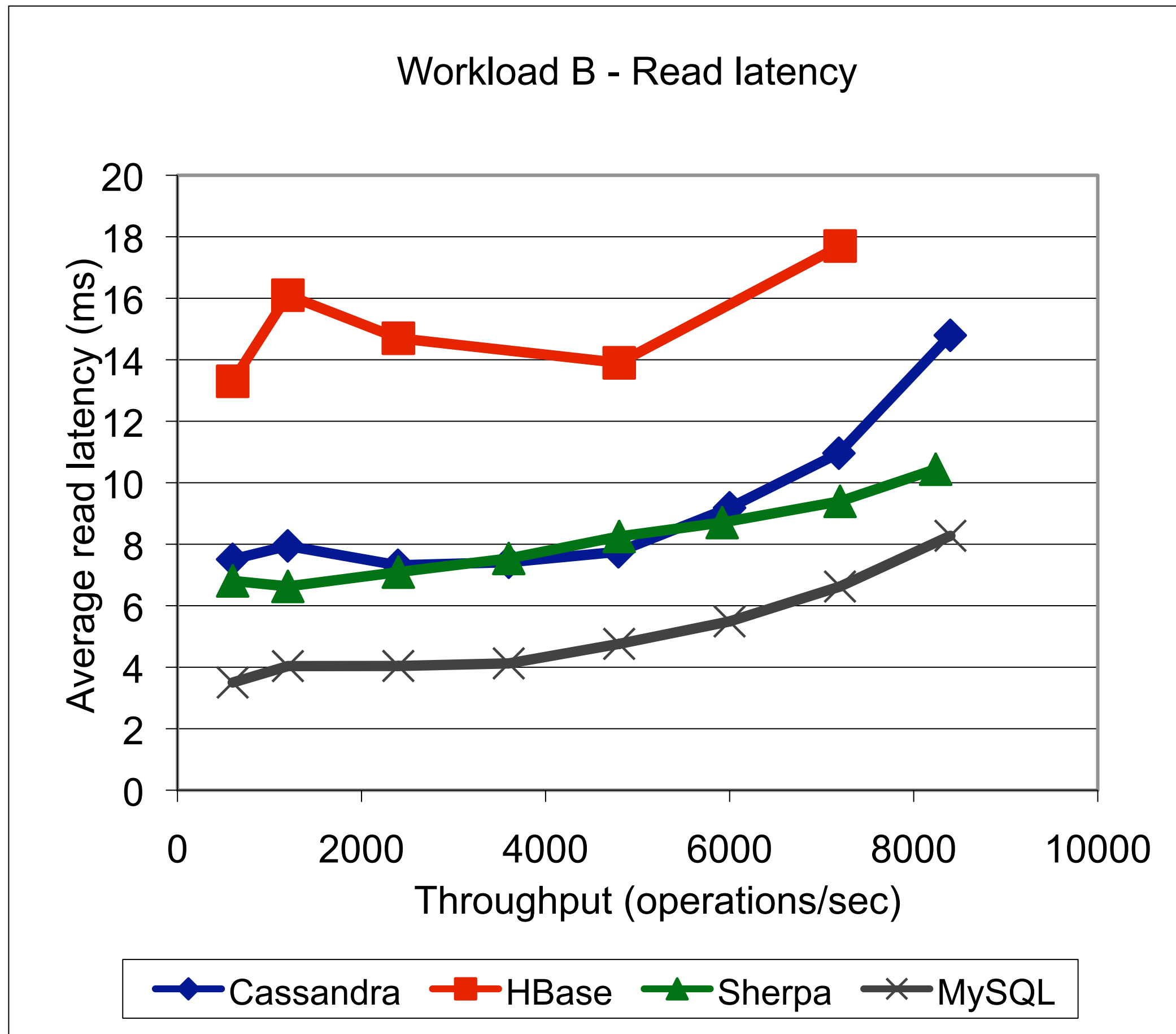
# Full Ehcache Storage Design



# Full Ehcache Storage Design

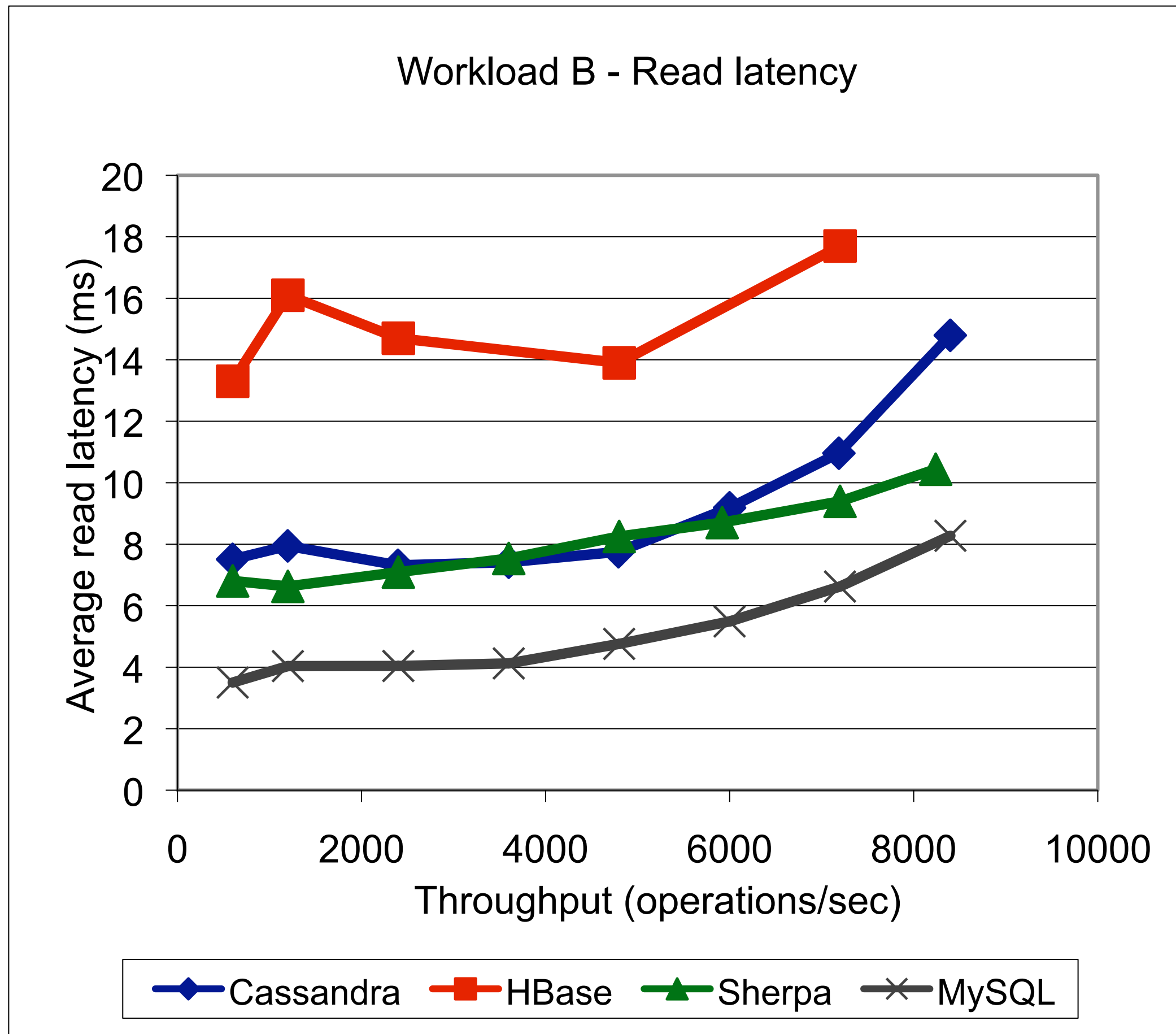


# Comparative Speeds



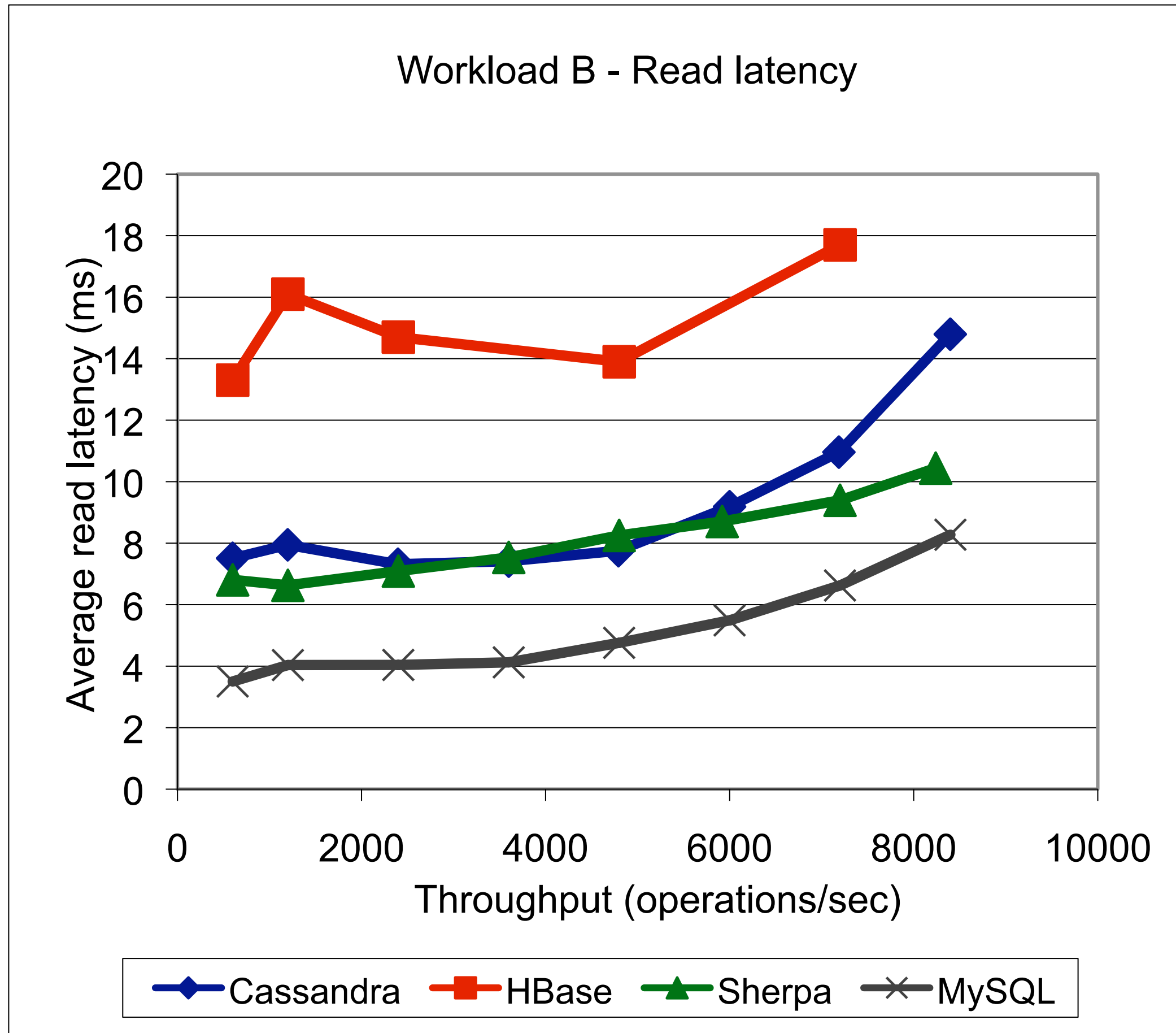
The code is available publicly on GitHub: <https://github.com/brianfrankcooper/YCSB>

# Comparative Speeds



The code is available publicly on GitHub: <https://github.com/brianfrankcooper/YCSB>

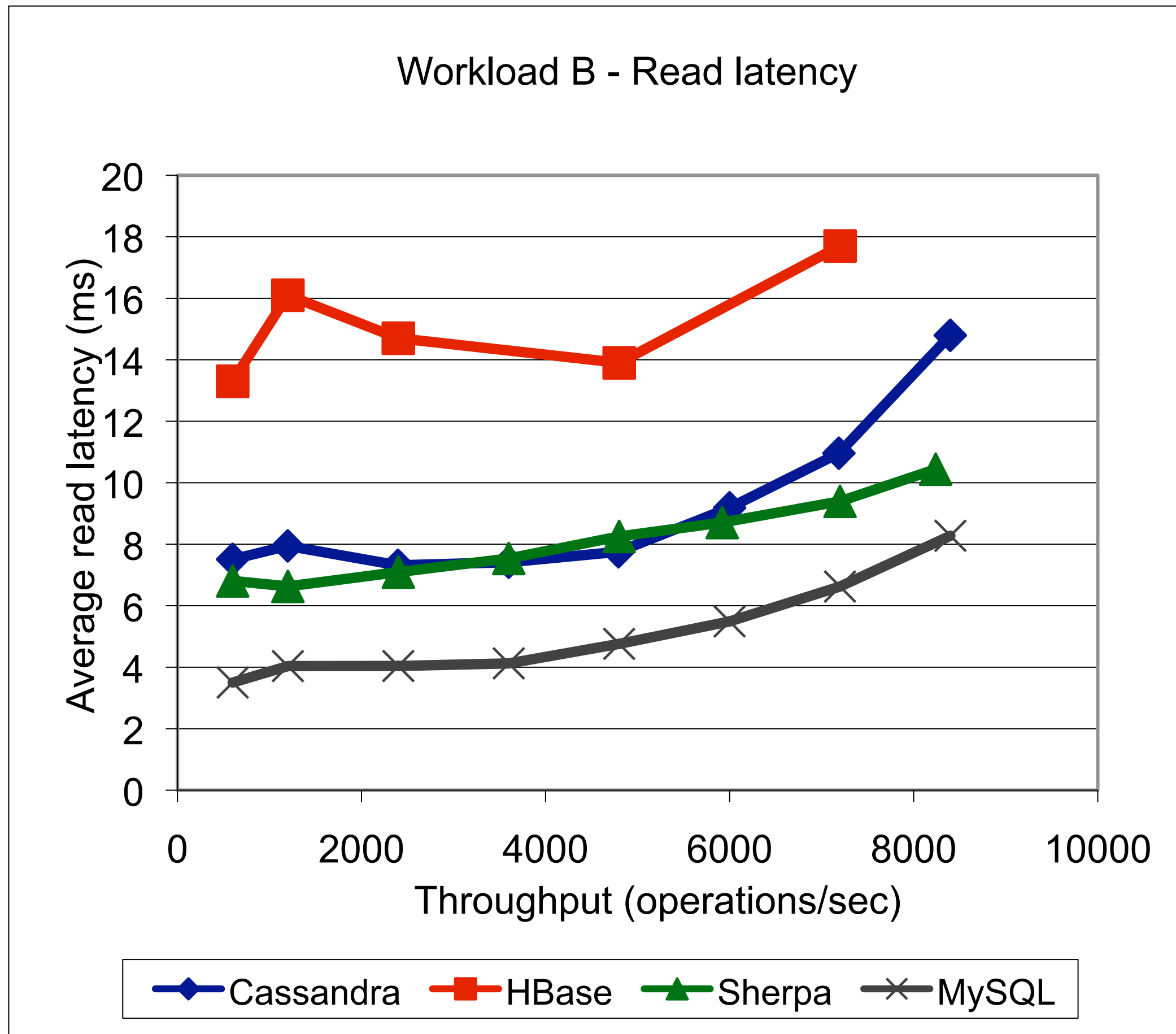
# Comparative Speeds



Compared with hybrid in-process and distributed cache:

$$\text{Latency} = \text{L1 speed} * \text{proportion} + \text{L2 speed} * \text{proportion}$$

# Comparative Speeds



Compared with hybrid in-process and distributed cache:

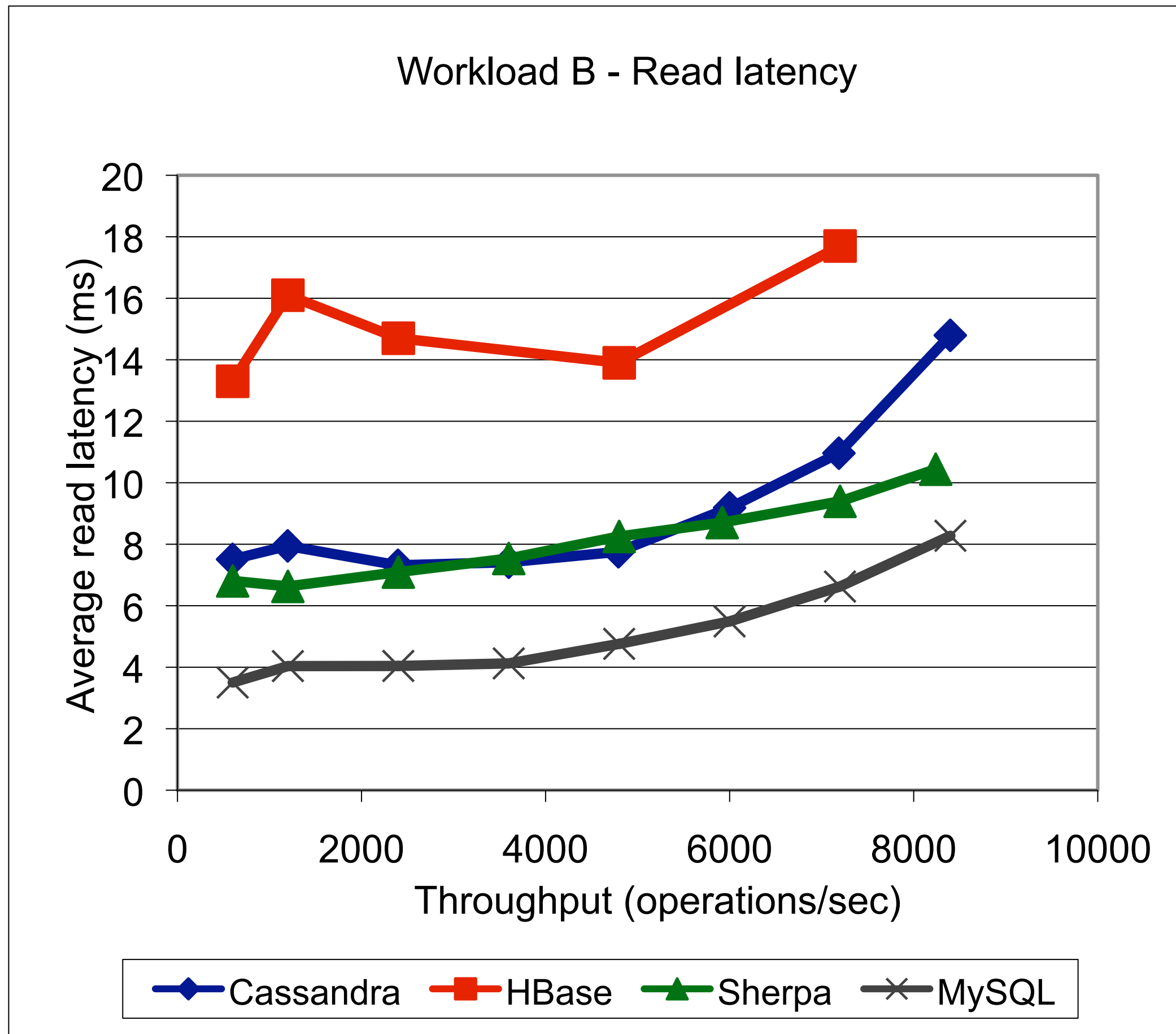
$$\text{Latency} = \text{L1 speed} * \text{proportion} + \text{L2 speed} * \text{proportion}$$

L1 = 0ms (< 5us) for on-heap and 50-100 us off-heap  
L2 = 2-3ms

80% L1 Pareto Model:

$$= 0 * .8 + 3 * .2$$
$$= .6 \text{ ms}$$

# Comparative Speeds



Compared with hybrid in-process and distributed cache:

$$\text{Latency} = \text{L1 speed} * \text{proportion} + \text{L2 speed} * \text{proportion}$$

L1 = 0ms (< 5us) for on-heap and 50-100 us off-heap  
L2 = 2-3ms

80% L1 Pareto Model:

$$= 0 * .8 + 3 * .2$$
$$= .6 \text{ ms}$$

90% L1 Pareto Model:

$$\text{latency} = 0 * .9 + 3 * .1$$
$$= .3 \text{ ms}$$

# Caching Toolkit

# Maximising Cache Efficiency

cache efficiency = cache hits / total hits

- ➔ High efficiency = high offload
- ➔ High efficiency = high performance

# Coherency with the SOR

- Usually handled with TTL

## Better Patterns

### **Eternal + Invalidation Pattern**

Cache elements are eternal. Elements are evicted from all caches when a change is made to the SOR.

Ehcache Implementations:

`cache.remove()` in-process

HTTP Delete using REST API via Cache Server

### **Write-Through Pattern**

The SOR gets updated in-line with the cache.

Ehcache Implementations:

- Hibernate read-write and transactional strategies.
- Ehcache CacheWriters

# Coherency with the SOR

## Better Patterns

### **Eternal + Invalidation Pattern**

Cache elements are eternal. Elements are evicted from all caches when a change is made to the SOR.

Ehcache Implementations:

`cache.remove()` in-process

HTTP Delete using REST API via Cache Server

### **Write-Through Pattern**

The SOR gets updated in-line with the cache.

Ehcache Implementations:

- Hibernate read-write and transactional strategies.
- Ehcache CacheWriters

# Why run an application cluster?

$n+1$

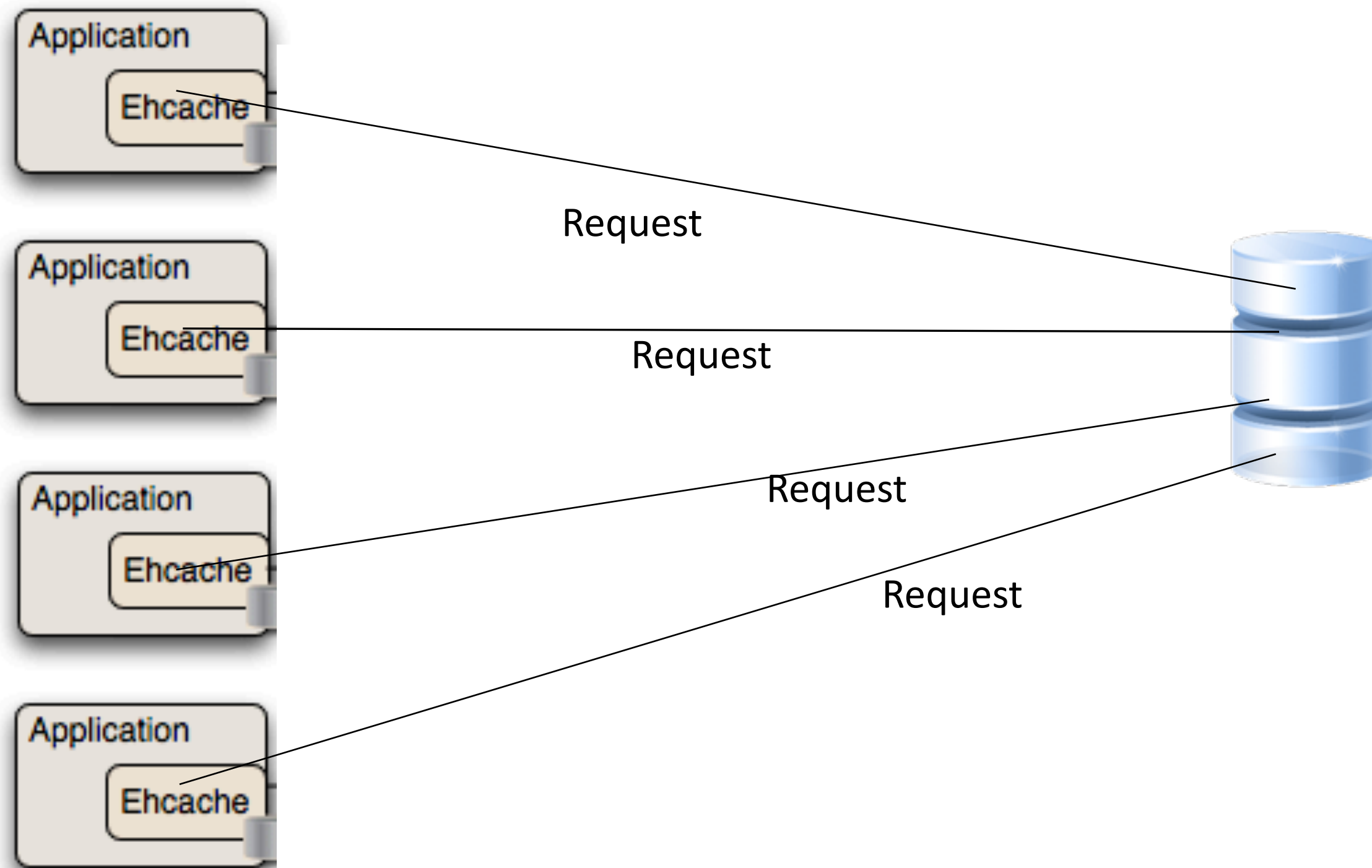
# Why run an application cluster?

$n+1$

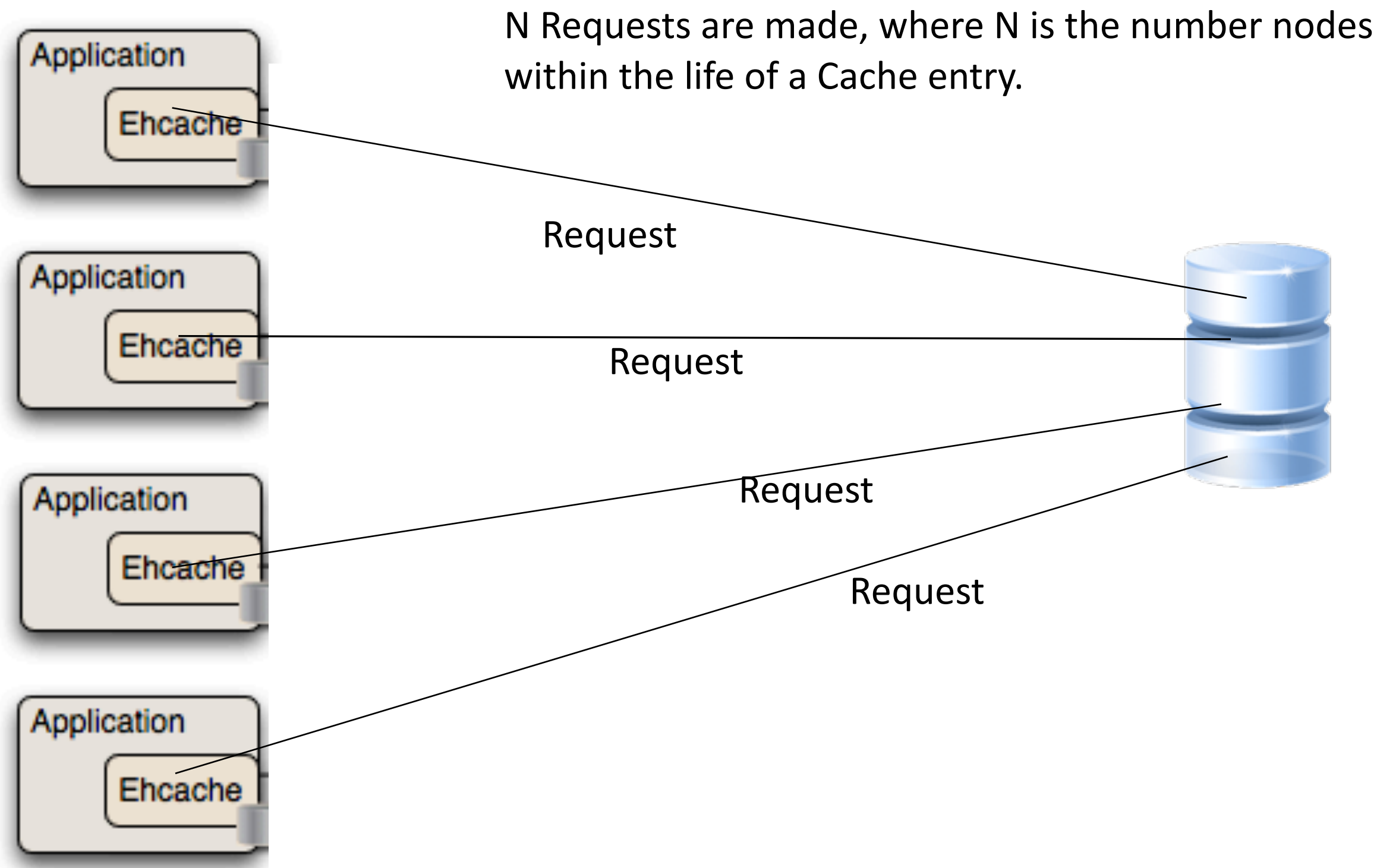
But this creates a new caching problems:

- N \* problem
- Bootstrap Problem
- Cluster coherency problem
- CAP theorem limits

# Standalone In-process Caching

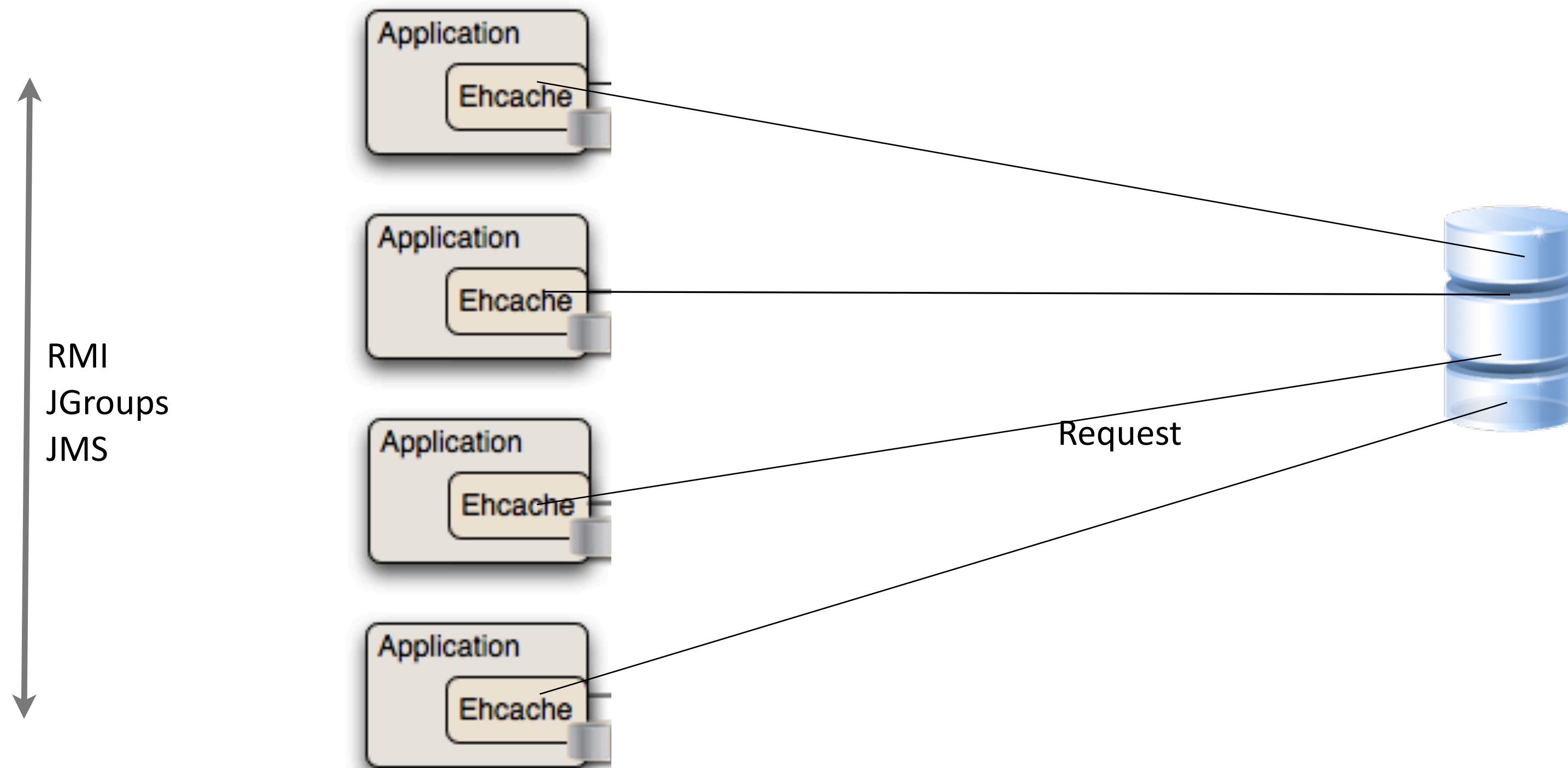


# Standalone In-process Caching

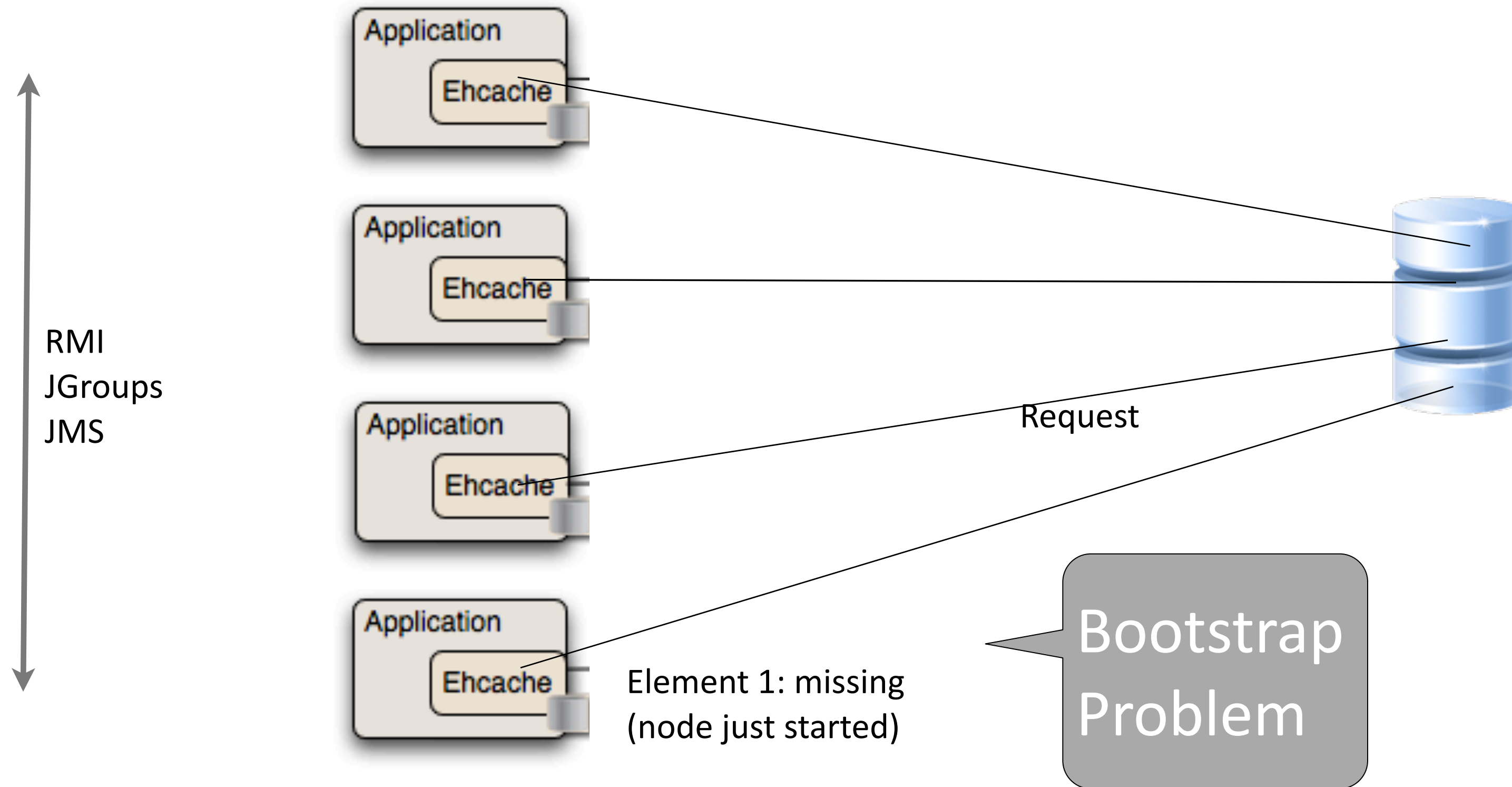


$N * \text{problem}$

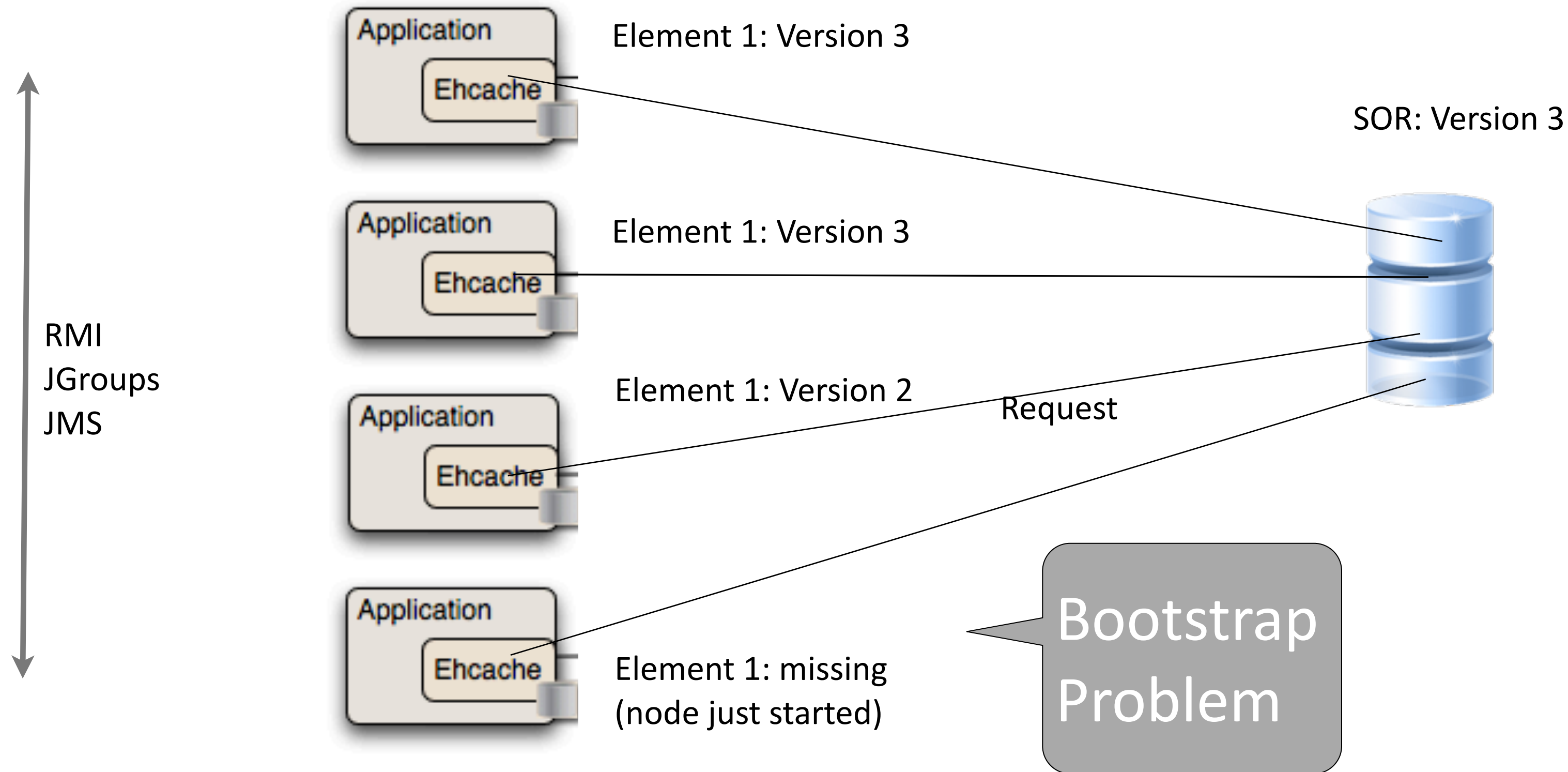
# Replicated in-process caching



# Replicated in-process caching

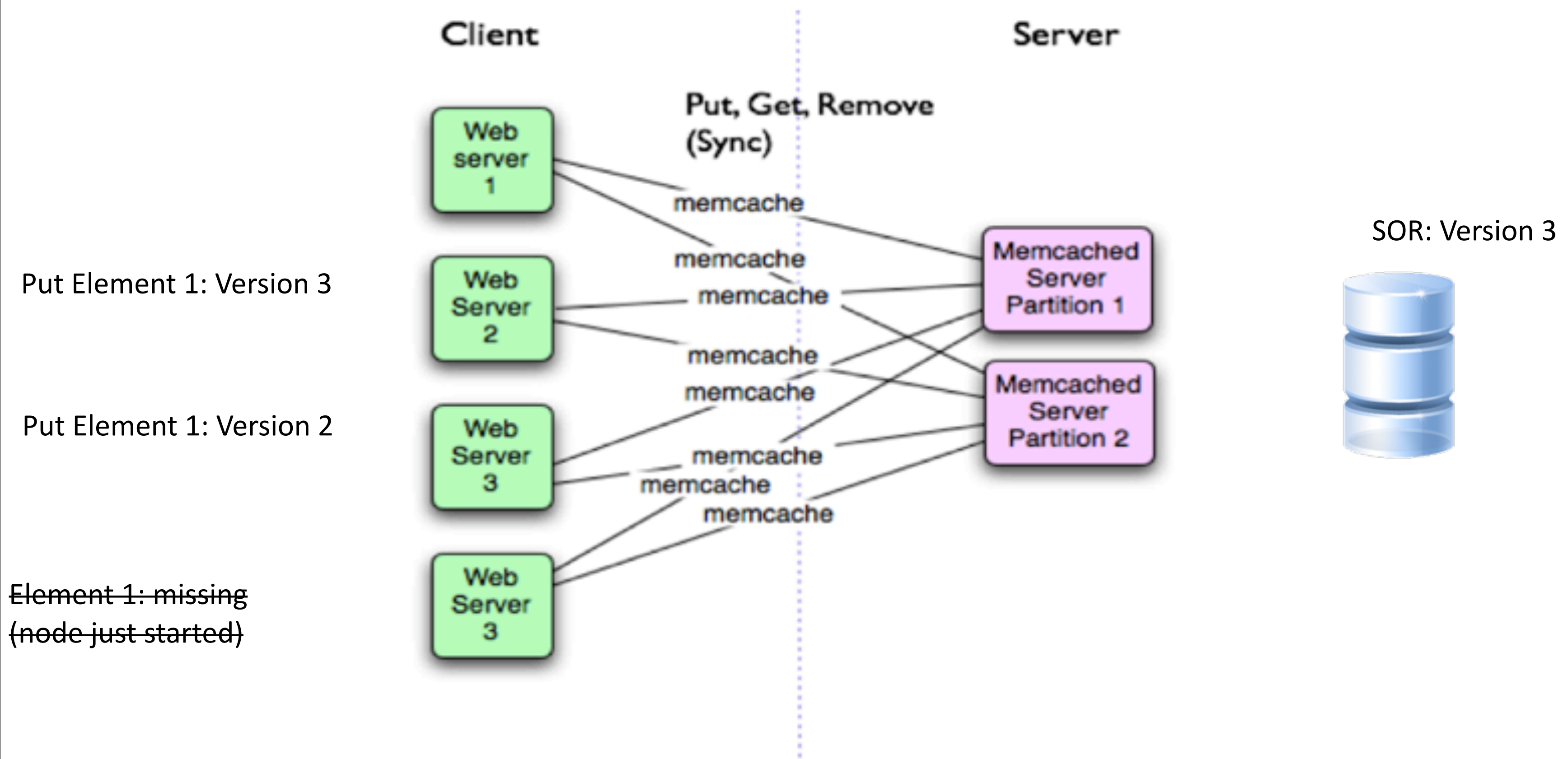


# Replicated in-process caching



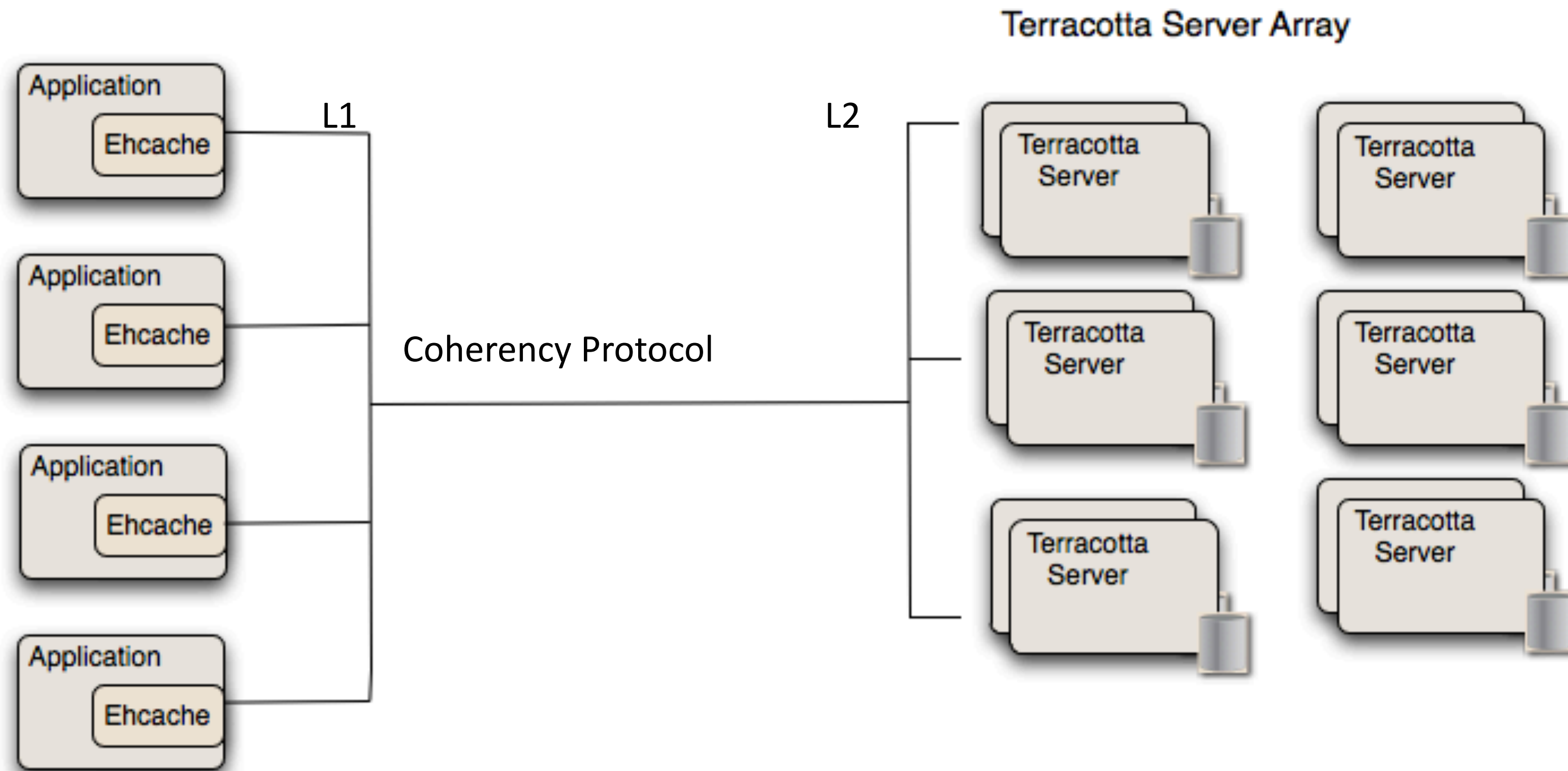
## Cache Coherency Problem

# Distributed Caching - memcache



## Cache Coherency Problem

# Strong Consistency Distributed Caching



# But... CAP Theorem

The CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- consistency,
- availability, and
- tolerance to partition.

# PACELC

**P**artition:

**A**vailability &

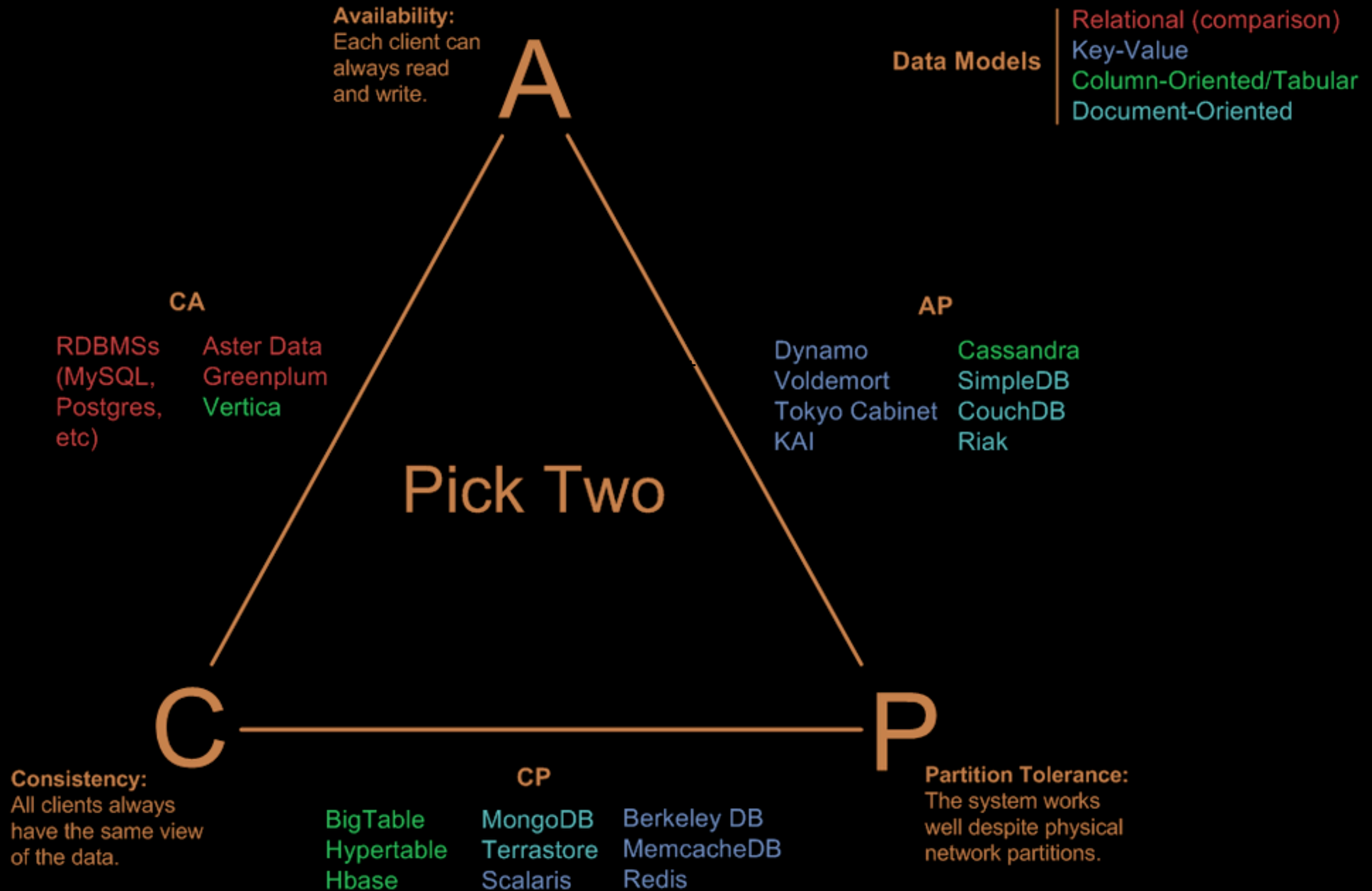
**C**onsistency

**E**lse:

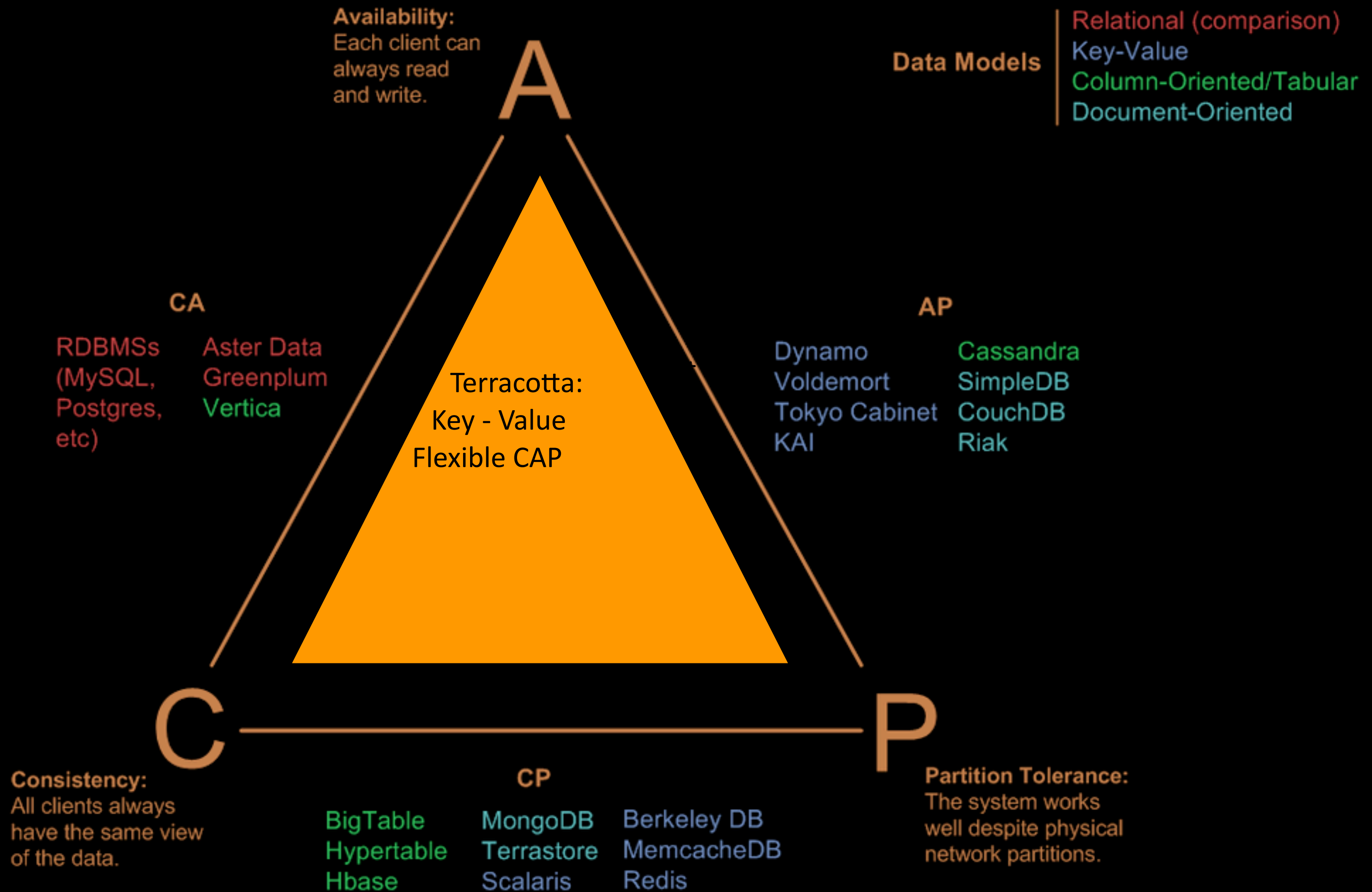
**L**atency &

**C**onsistency

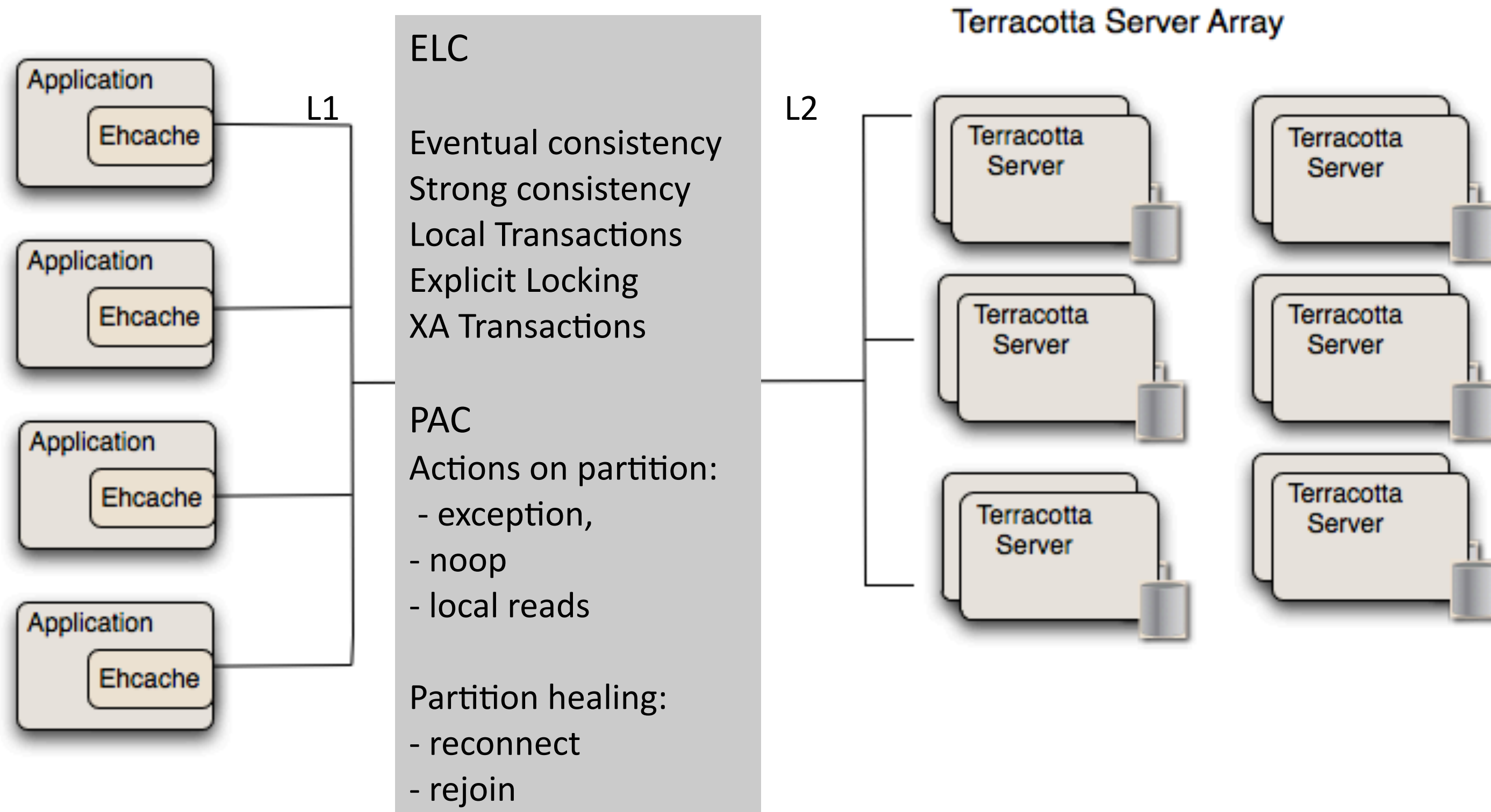
# Visual Guide to NoSQL Systems



# Visual Guide to NoSQL Systems



# CAP Trade-offs in Ehcache



# Additional Information

- Additional Slides showing applied areas with Ehcache
- Project Website: [www.ehcache.org](http://www.ehcache.org)
- Documentation: [www.ehcache.org/documentation](http://www.ehcache.org/documentation)
- BigMemory: [www.terracotta.org/bigmemory](http://www.terracotta.org/bigmemory)
- Commercial Version: [www.terracotta.org/ehcache/](http://www.terracotta.org/ehcache/)
- Twitter: ehcache and gregrluck

# Applied Solutions with Ehcache

.

# ehcache.xml for standalone

ehcache.xml:

```
<ehcache>
  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToLiveSeconds="120"
  />

  <cache name="org.hibernate.cache.UpdateTimestampsCache"
    maxElementsInMemory="10000"
    timeToIdleSeconds="300"
  />

  <cache name="org.hibernate.cache.StandardQueryCache"
    maxElementsInMemory="10000"
    timeToIdleSeconds="300"
  />
</ehcache>
```

# ehcache.xml for distributed caching

## ehcache.xml

```
<ehcache>
  <terracottaConfig url="someserver:9510" />
  <defaultCache
    maxElementsInMemory="10000"
    eternal="false"
    timeToLiveSeconds="120"
  />
  <cache name="com.company.domain.Pets"
    maxElementsInMemory="10000"
    eternal="true">
    <terracotta clustered="true" coherent="false" />
  </cache>
  <cache name="com.company.domain.Pets"
    maxElementsInMemory="10000"
    timeToLiveSeconds="3000">
    <terracotta clustered="true" coherent="true" />
  </cache>
</ehcache>
```

# Maven Interactive Use

- Starting the Terracotta Server

```
mvn tc:start
```

- Stopping the Terracotta Server

```
mvn tc:stop
```

# Maven integration testing

```
<build>
  <plugins>
    <plugin>
      <groupId>org.terracotta.maven.plugins</groupId>
      <artifactId>tc-maven-plugin</artifactId>
      <version>1.5.1</version>
      <executions>
        <execution>
          <id>run-integration</id>
          <phase>pre-integration-test</phase>
          <goals>
            <goal>run-integration</goal>
          </goals>
        </execution>
        <execution>
          <id>terminate-integration</id>
          <phase>post-integration-test</phase>
          <goals>
            <goal>terminate-integration</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

# Ant-based Builds

- See UsingWithAnt.txt in the terracotta directory of the ehcache-2.1 distribution
- Uses a macrodef which integrates with Maven
- Sample start and stop targets for interactive use or dependent targets for integration tests
- Demo

# Applications: Web tier caching

.

# Web Caching

- Ehcache provides CachingFilter and PageFragmentCachingFilter
- Use “Simple” versions or subclass for greater control
- Cache anything that you can generate in a web container: html, XML, JSON, images, binary files ...
- Very high performance.
- Excellent clustering characteristics. Responses are gzipped and already mostly byte[].
- Scalable due to BlockingCache
- Tested on all major web containers and app servers
- Distributed Caching as usual with Terracotta



Example Web Page

# Web Caching Steps

- Use or Subclass SimpleCachingFilter or SimplePageFragmentCachingFilter
- Configure a filter in web.xml
- Create an URL mapping in web.xml
- Configure a cache entry matching the filter name

# Configure web.xml

```
<filter>
  <filter-name>SimplePageCachingFilter</filter-name>
  <filter-class>net.sf.ehcache.constructs.
    web.filter.SimplePageCachingFilter
</filter-class>
</filter>
<filter-mapping>
  <filter-name>SimplePageCachingFilter</filter-name>
  <url-pattern>/index.jsp</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

# Create a cache in ehcache.xml

```
<cache name="SimplePageCachingFilter"  
  maxElementsInMemory="10000"  
  maxElementsOnDisk="1000"  
  eternal="false"  
  overflowToDisk="true"  
  timeToIdleSeconds="300"  
  timeToLiveSeconds="600"  
  memoryStoreEvictionPolicy="LFU"  
>
```

# Web Page Speedup

- Say a web page takes 1 second and Distributed Ehcache can retrieve a page from either its L1 or L2 for via the SimplePageCachingFilter in an average of 1 ms. Also assume, the browser is on the LAN.
- Because the web page is the end result of a computation, it has a proportion of 100%.
- The expected system speedup is thus:
$$1 / ((1 - 1) + 1 / 1000)$$
$$= 1 / (0 + .001)$$
$$= 1000 \text{ times system speedup}$$

# Applications: Database caching

.

# Hibernate, OpenJPA and DAO

- Hibernate 3.3+ Caching SPI

- Old SPI was heavily synchronized and not well suited to clusters
- New SPI uses CacheRegionFactory
- Fully cluster safe with Terracotta Server Array
- Unification of the Ehcache and Terracotta 3.2 providers
- All strategies including <transactional>

- OpenJPA

- 1 and 2
- Fully cluster safe with Terracotta Server Array

- DAO

- Uses the general API
- See <http://ehcache.org/documentation/jdbc.html>

# Database Speedup

- A Hibernate Session.load() for a single object is about 1000 times faster from cache than from a database.
- A typical Hibernate query will return a list of IDs from the database, and then attempt to load each. If Session.iterate() is used Hibernate goes back to the database to load each object.
- Imagine a scenario where we execute a query against the database which returns a hundred IDs and then load each one.
- The query takes 20% of the time and the roundtrip loading takes the rest (80%). The database query itself is 75% of the time that the operation takes. The proportion being sped up is thus 60% (75% \* 80%).
- The expected system speedup is thus:

$$\begin{aligned} & 1 / ((1 - .6) + .6 / 1000) \\ & = 1 / (.4 + .006) \\ & = 2.5 \text{ times system speedup} \end{aligned}$$

# Applications: General Caching

.

# General API

- Use in your own application code
- Create your caches in ehcache.xml or programmatically
- Very rich API
- Simple example:

```
CacheManager manager = new CacheManager();  
Ehcache cache1 = manager.getCache("cache1");  
cache1.put(new Element("key1", "value"));  
Element element1 = cache1.get("key1");
```

# General Caching Speedup

- A system calls a REST service in France from Australia. The latency is 200ms each way just due to the speed of light in a fibre. The entire REST call, including XML transformations at each end is 3 seconds. Further processing creates the web page in another 1 second.
- A distributed cache is being used with good locality of reference. The average cache retrieval time is .5 ms.
- The expected system speedup is thus:

$$\begin{aligned} & 1 / ((1 - .75) + .75 / 2000) \\ & = 1 / (.25 + .000375) \\ & = 3.99 \text{ times system speedup} \end{aligned}$$

# Ehcache Console

- Web based
- Configuration
- Efficiency
- Memory Use

**EHCACHE Console**  
a product from **TERRACOTTA**

Cache Managers Statistics Configuration Contents Cache Efficiency Memory Use API

Efficiency for cache manager [392.358.1.48-50051.CacheManager@2b65340c] and cache [o.t.s.r.ip.t.Charles Towers]



**EHCACHE Console**  
a product from **TERRACOTTA**

[Documentation | Licensing & Support](#)

No license key detected

Cache Managers Statistics Configuration Contents Cache Efficiency Memory Use API

Statistics for cache manager [All] [Settings](#)

Cache	Cache Instan...	Total Elements	Hit Ratio	Misses	Misses/s	Put/s
Albury	1	7	0.0%	0.0	0.2	0.2
Freemantle	1	25	0.0%	0.0	0.2	0.2
Oldstone	1	19	0.0%	0.0	0.2	0.2
Ligon-Cly	1	20	0.0%	0.0	0.2	0.2
Herbath	1	24	0.0%	0.0	0.2	0.2
org.terracotta.sonic.realty.org.package.name.tdtest.Mullingong	1	19	0.0%	0.0	0.2	0.2
Newcastle	1	29	0.0%	0.0	0.3	0.3
Wodonga	1	22	0.0%	0.0	0.3	0.3
org.terracotta.sonic.realty.org.package.name.tdtest.Perth	1	26	0.0%	0.0	0.3	0.3
<a href="#">Documentation   Licensing &amp; Support</a>	1	30	0.0%	0.0	0.4	0.4
No license key detected	1	30	0.0%	0.0	0.4	0.4
tdtest.Harsham	2	44	25.0%	0.1	0.3	0.4
tdtest.Harsham	1	44	0.0%	0.0	0.5	0.5
tdtest.Puckingham	1	25	0.0%	0.0	0.5	0.5
tdtest.Harsham	1	68	33.3%	0.3	0.6	0.8
tdtest.Geraldton	1	36	11.1%	0.1	0.6	0.9
tdtest.Charles ...	1	124	38.8%	0.7	1.1	1.6
tdtest.Oldstone	1	35	0.0%	0.0	1.6	1.6
tdtest.Dubbo	1	192	95.5%	0.4	0.3	4.0
tdtest.Mullingong	1	18	16.6%	1.1	0.5	7.2

- Included as a feature of commercial Ehcache versions
- API to connect Operations Monitoring