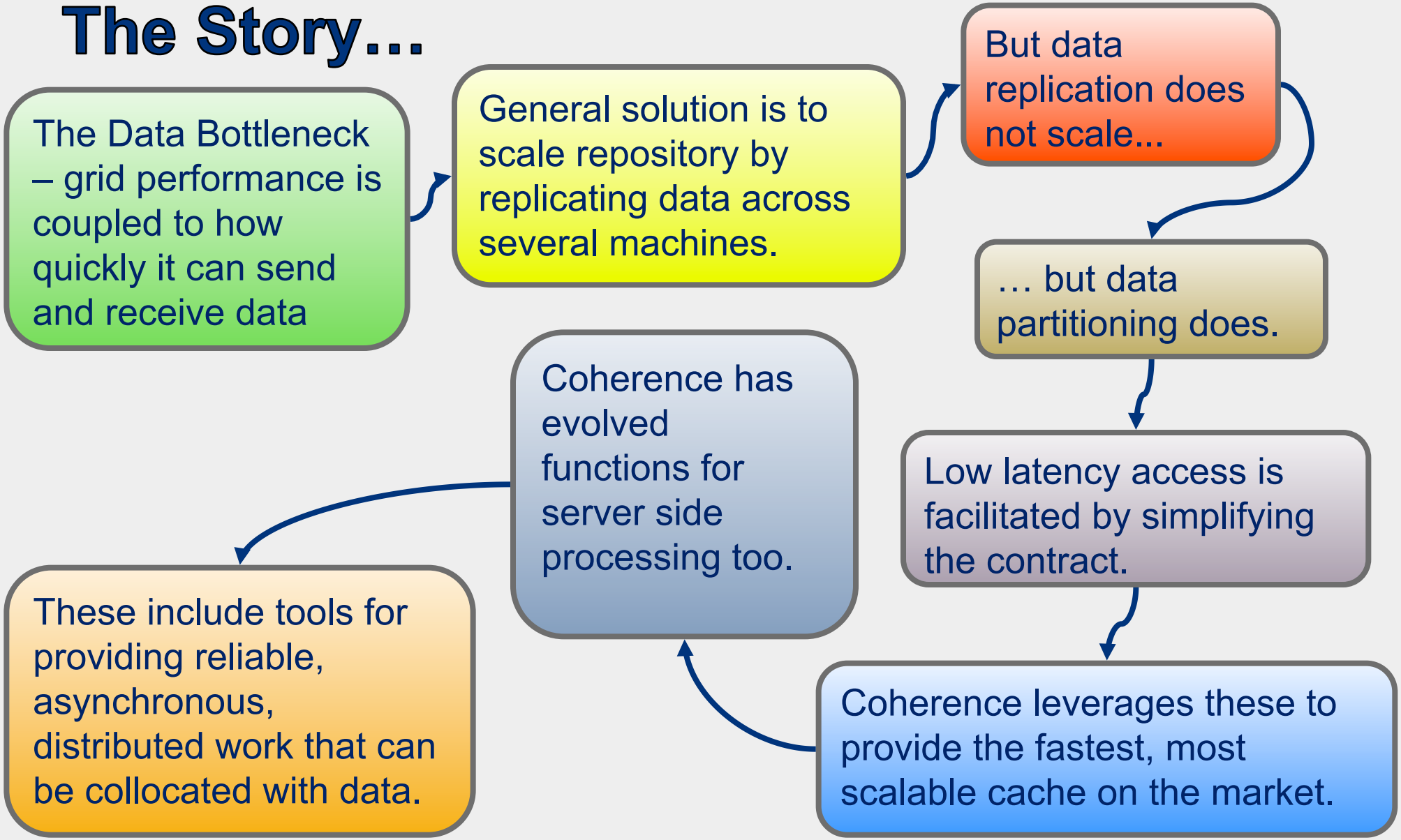
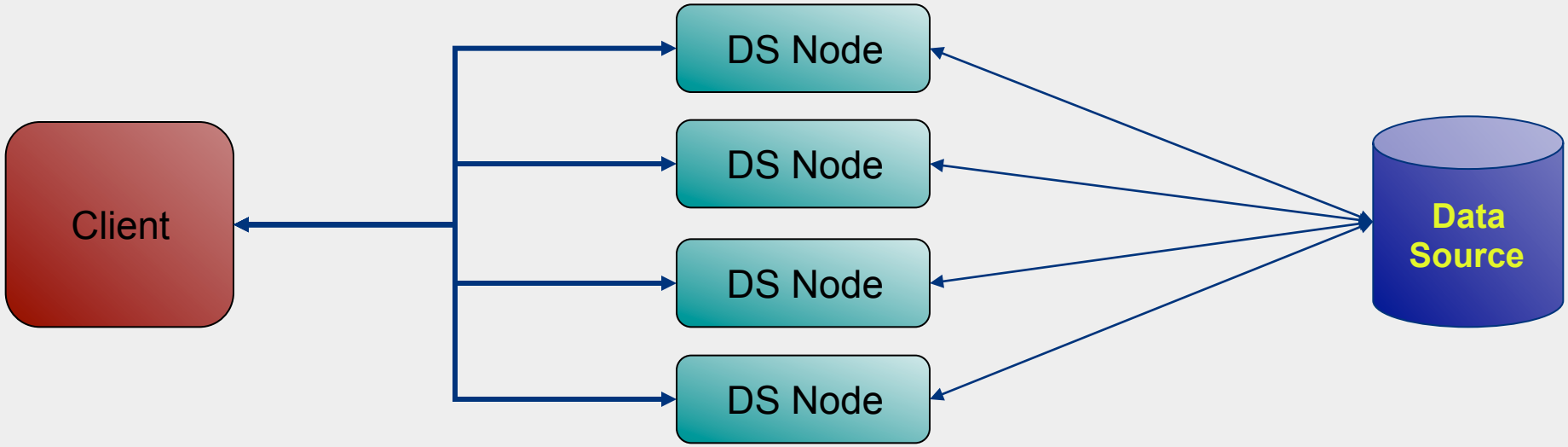
Data Clusters with Coherence

Benjamin Stopford

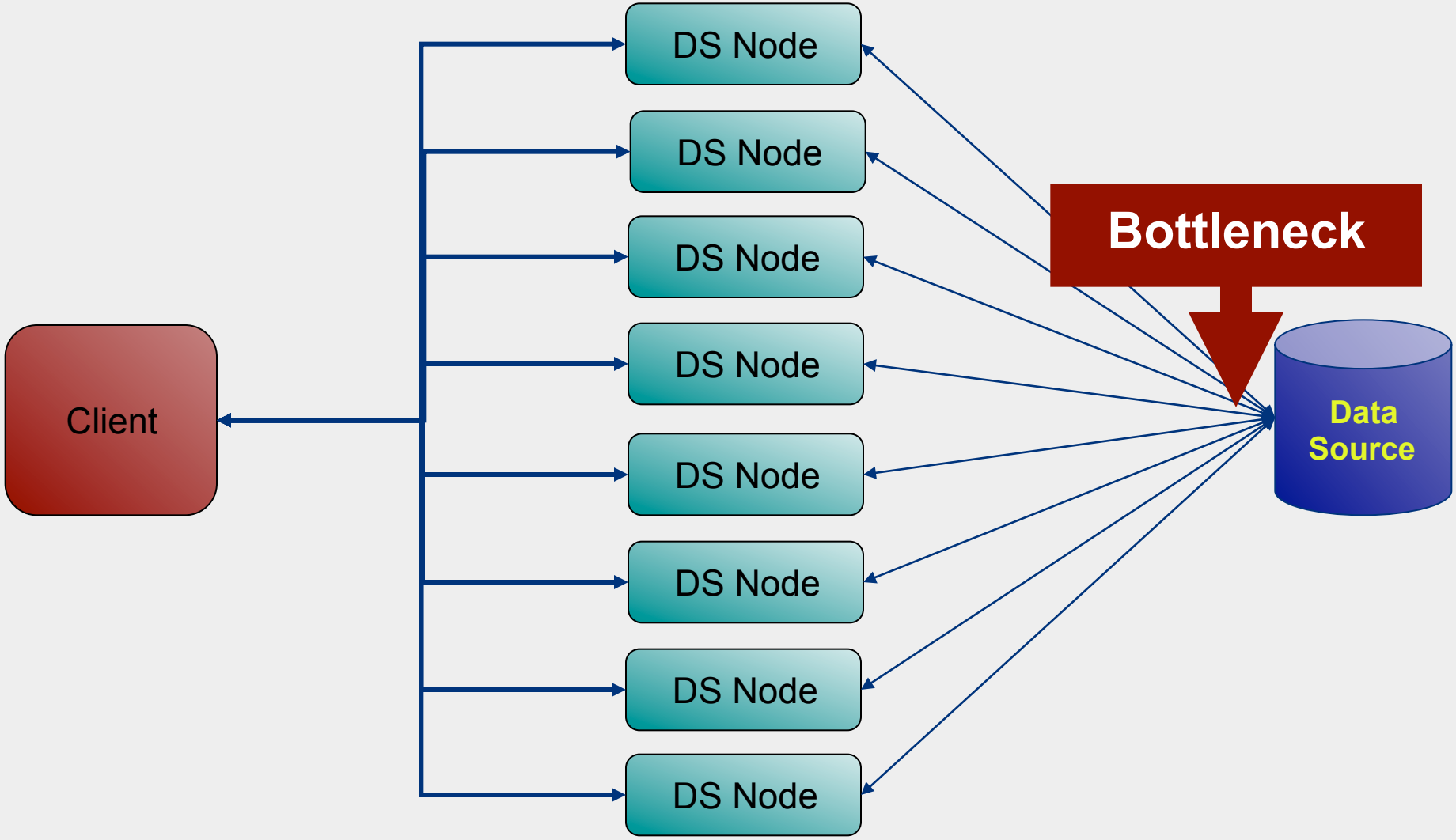
The Story…


The Data Bottleneck – grid performance is coupled to how quickly it can send and receive data
General solution is to scale repository by replicating data across several machines.
But data replication does not scale...
… but data partitioning does.
Low latency access is facilitated by simplifying the contract.
Coherence leverages these to provide the fastest, most scalable cache on the market.
Coherence has evolved functions for server side processing too.
These include tools for providing reliable, asynchronous, distributed work that can be collocated with data.

There is a problem with this architecture. What is it?

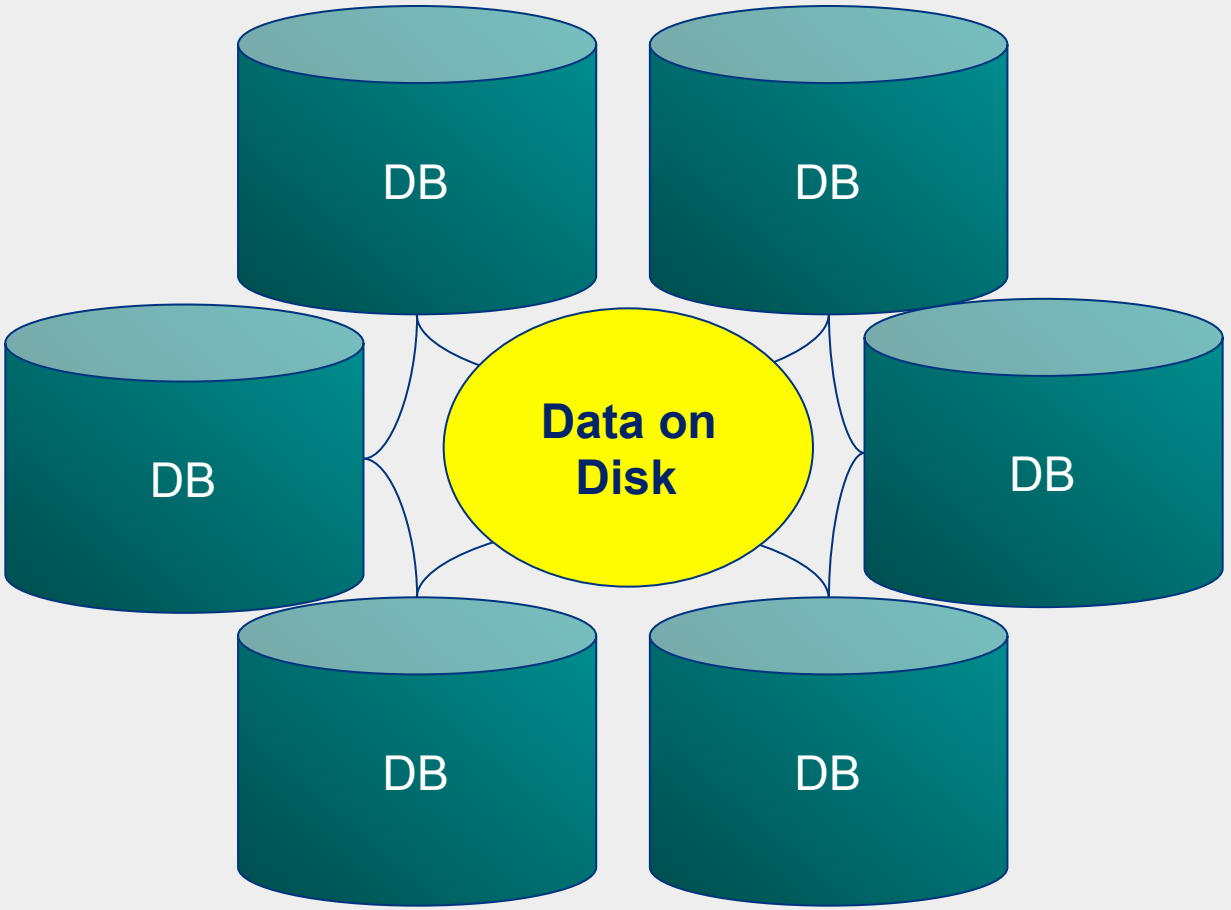


The database becomes a bottleneck as the grid scales out



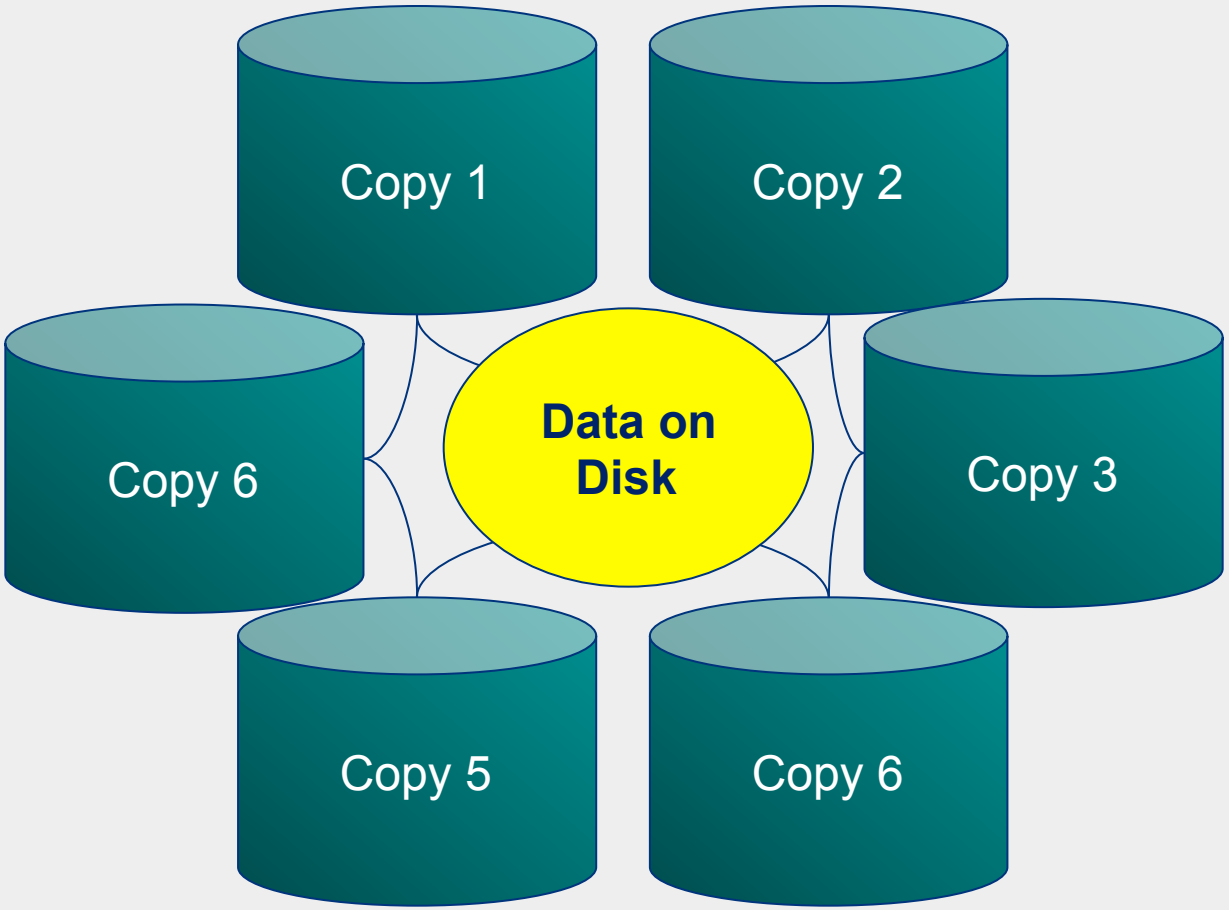
This is the Data Bottleneck Problem

Consider a Clustered Database – this is one solution

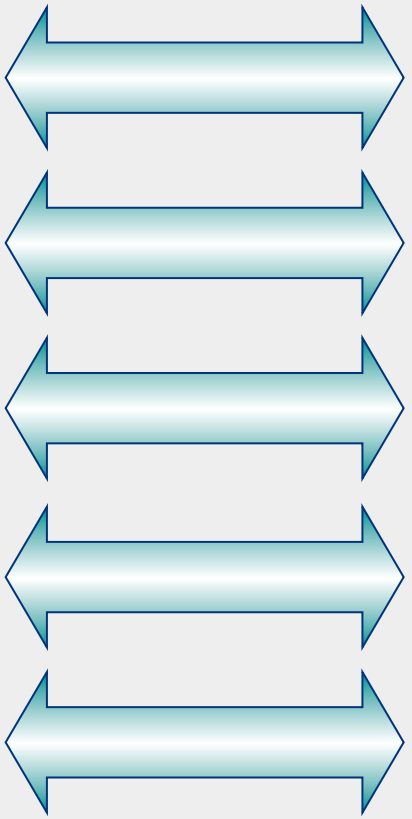


- Data exists on a shared file system.
- Multiple machines add bandwidth and processing ability

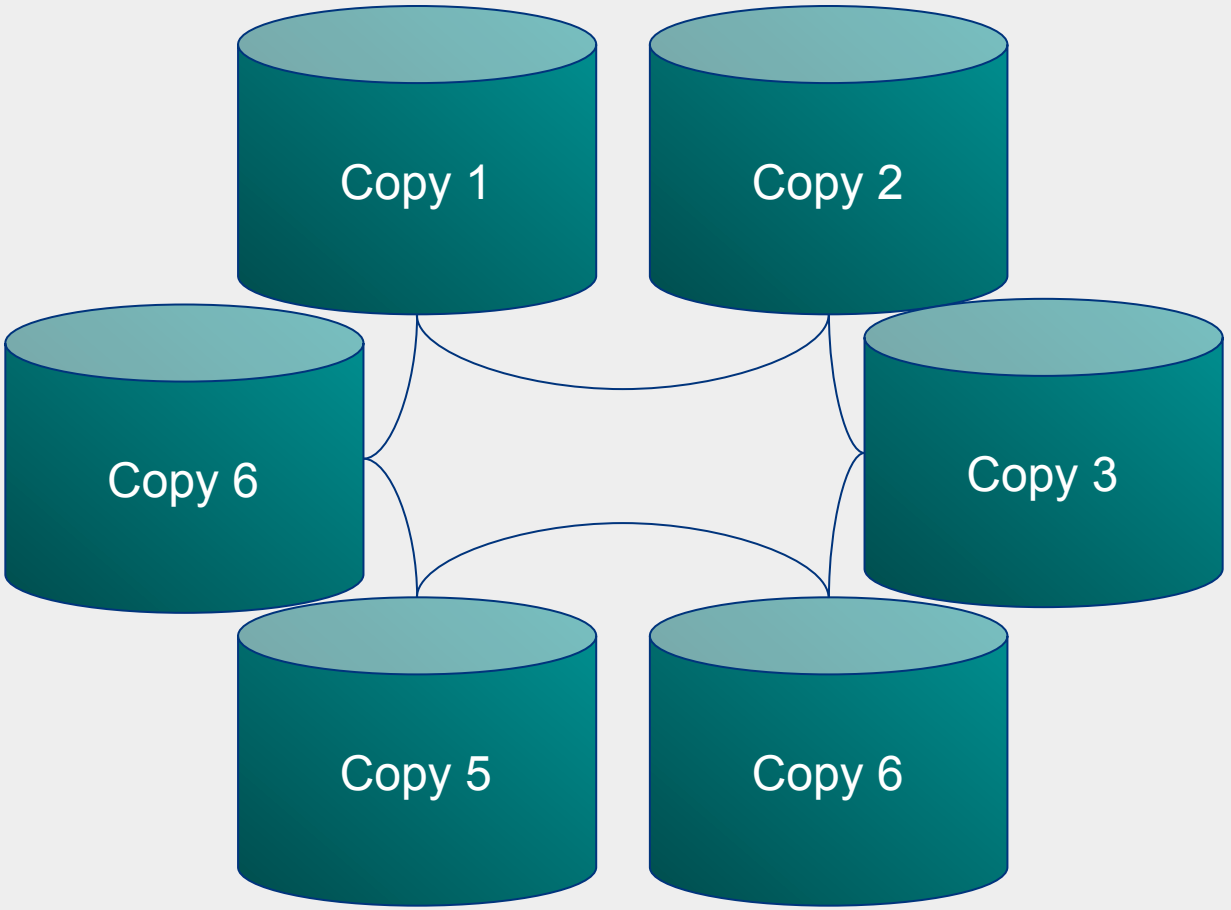
Data is cached in memory on each machine



Greatly increases bandwidth available for reading data.



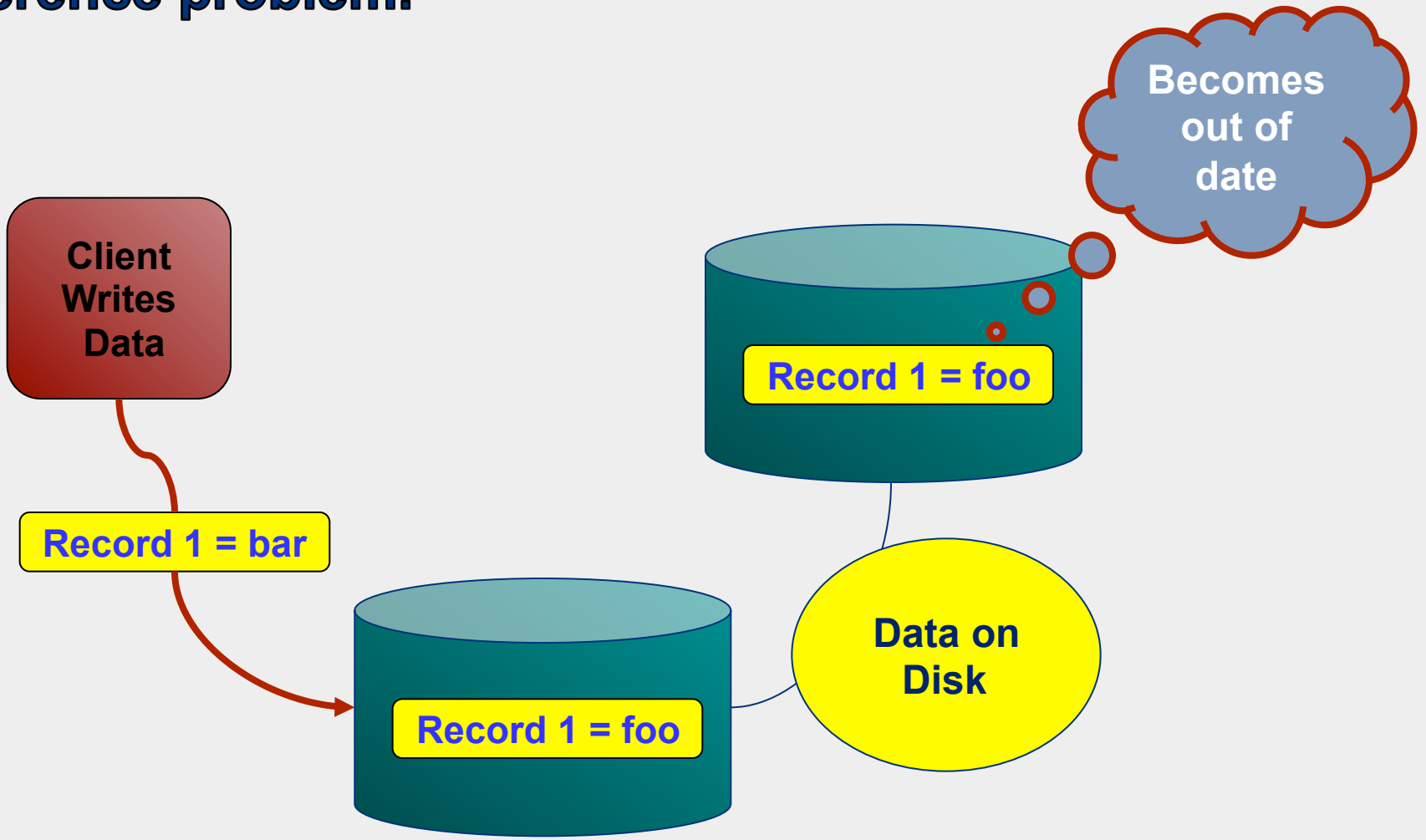
This architecture is that of a Replicated Cache



Data is copied to different machines.

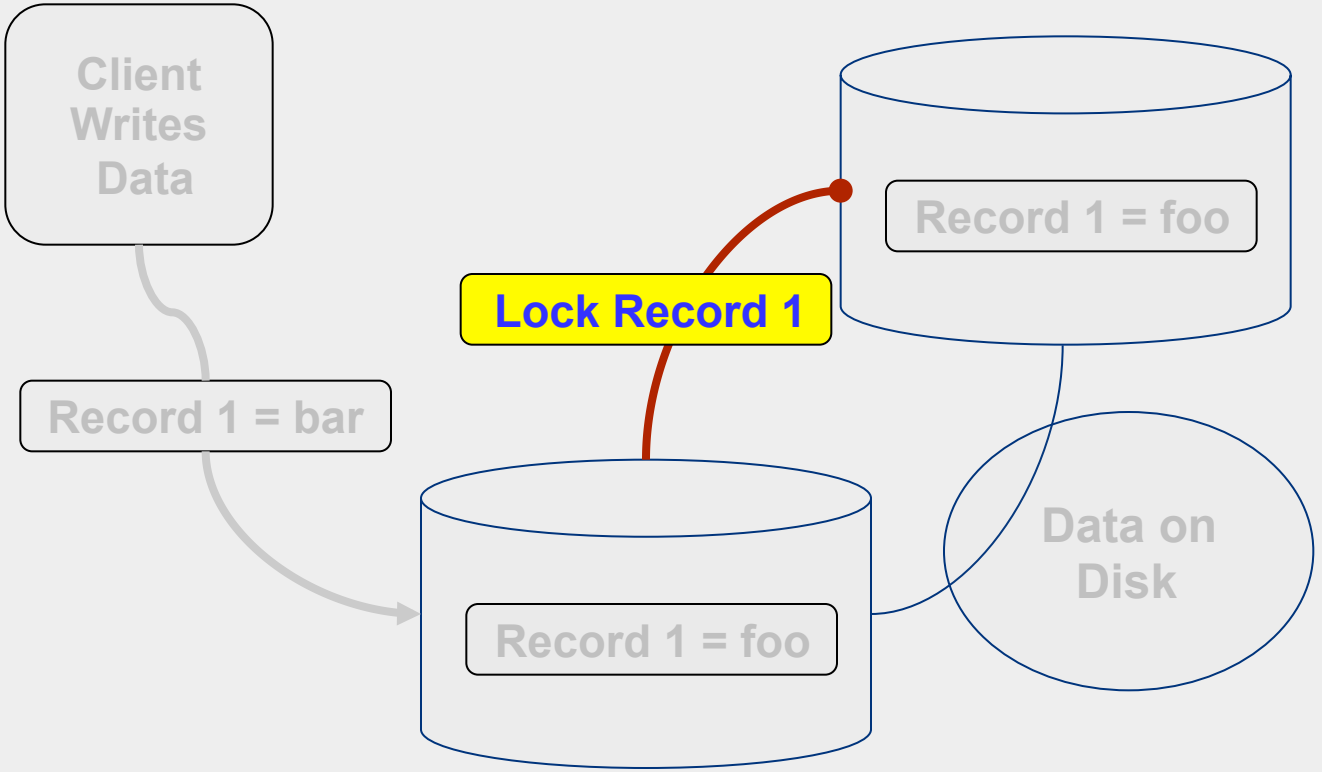
What is wrong with this architecture?

Multiple data copies must be kept in sync. There is a coherence problem.

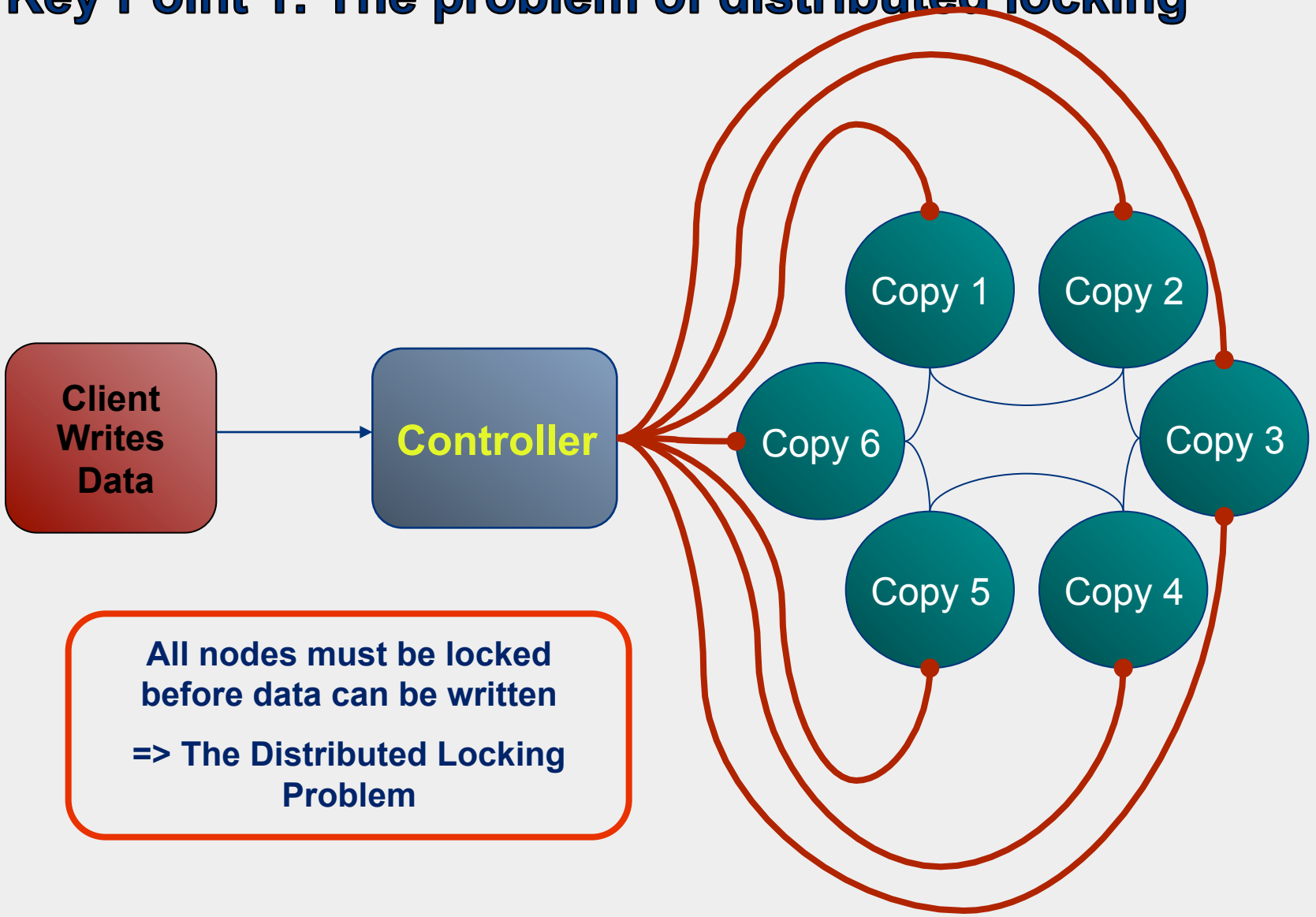


The writing of data leads to a distributed locking problem.

This does not scale.



Key Point 1: The problem of distributed locking

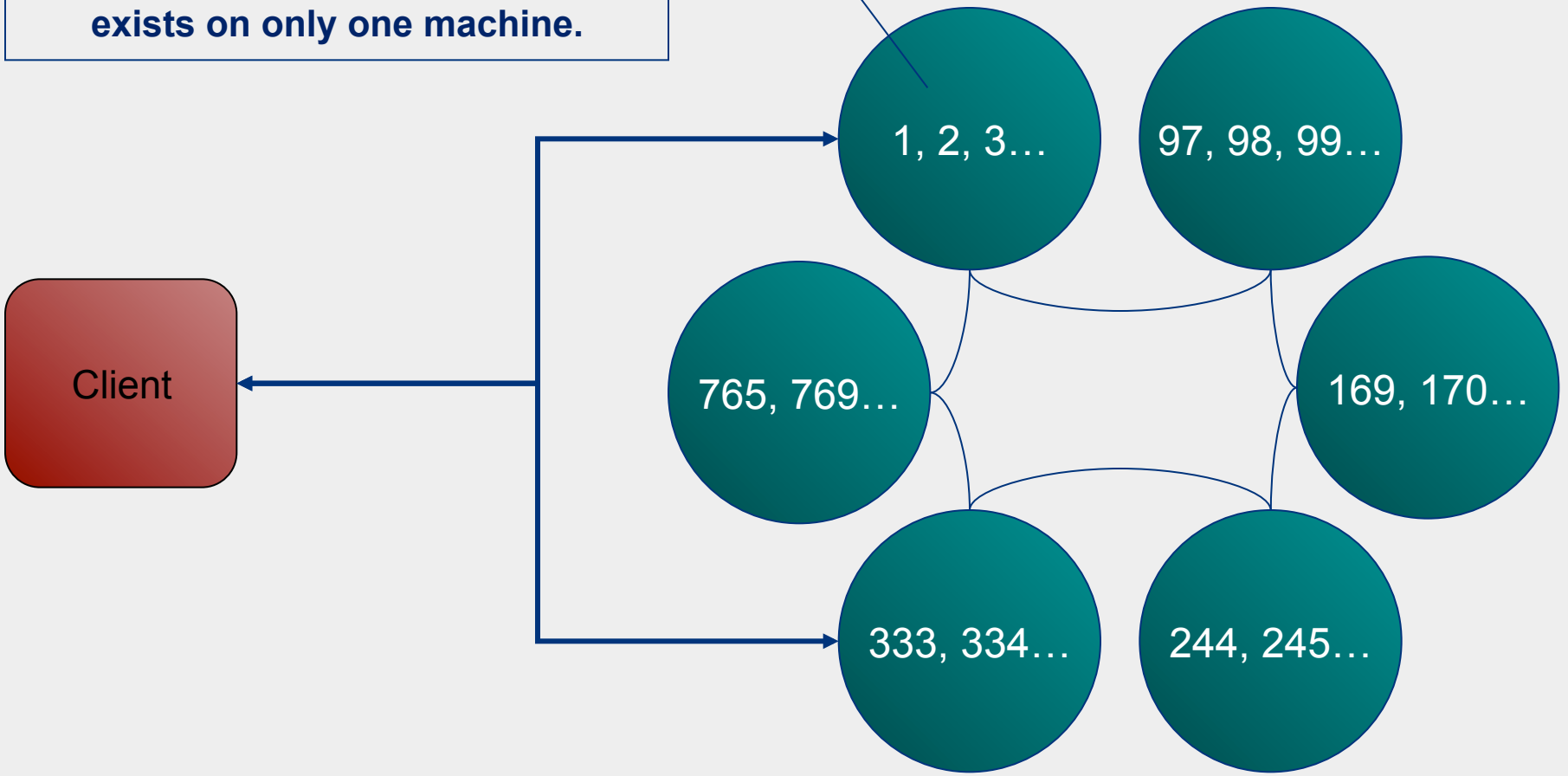


All nodes must be locked before data can be written

=> The Distributed Locking Problem

However there is an alternative: Data Partitioning

Each machine is responsible for a subset of the records. Each record exists on only one machine.



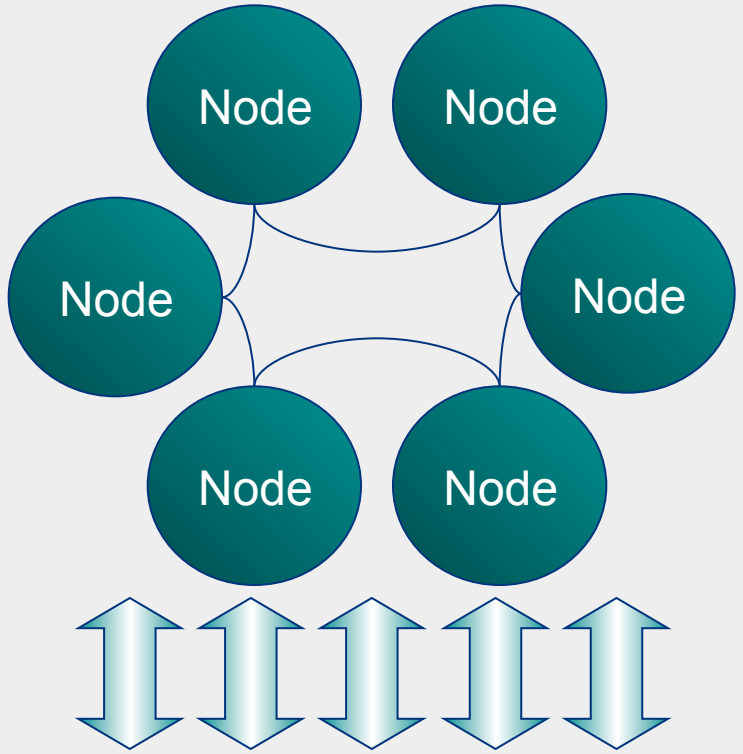
This solves both problems associated with the replicated architecture

- Data volume will naturally increase with the number of machines in the cluster as the data only exists in one place.
- Writes are only ever sent to one machine (that holds the singleton piece of data being modified) so write latency scales with the cluster.

So what technologies facilitate this process?

- Oracle RAC
- Gigaspaces
- Terracotta
- Oracle Coherence

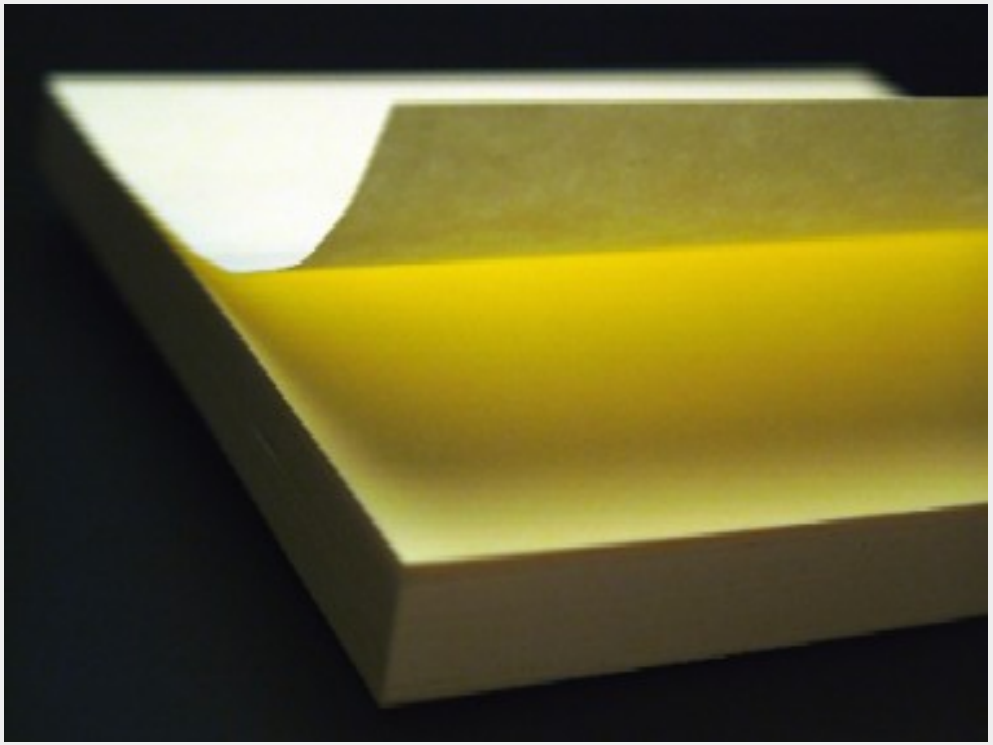
Introducing Coherence
A Story of Accidental Genius



In 1974 3M created a liquid tack. A glue that stuck but did not bond. This product endured fairly limited sales for some years until it was used to make a well known every day item.

What do you think that was?

The Post-It Note

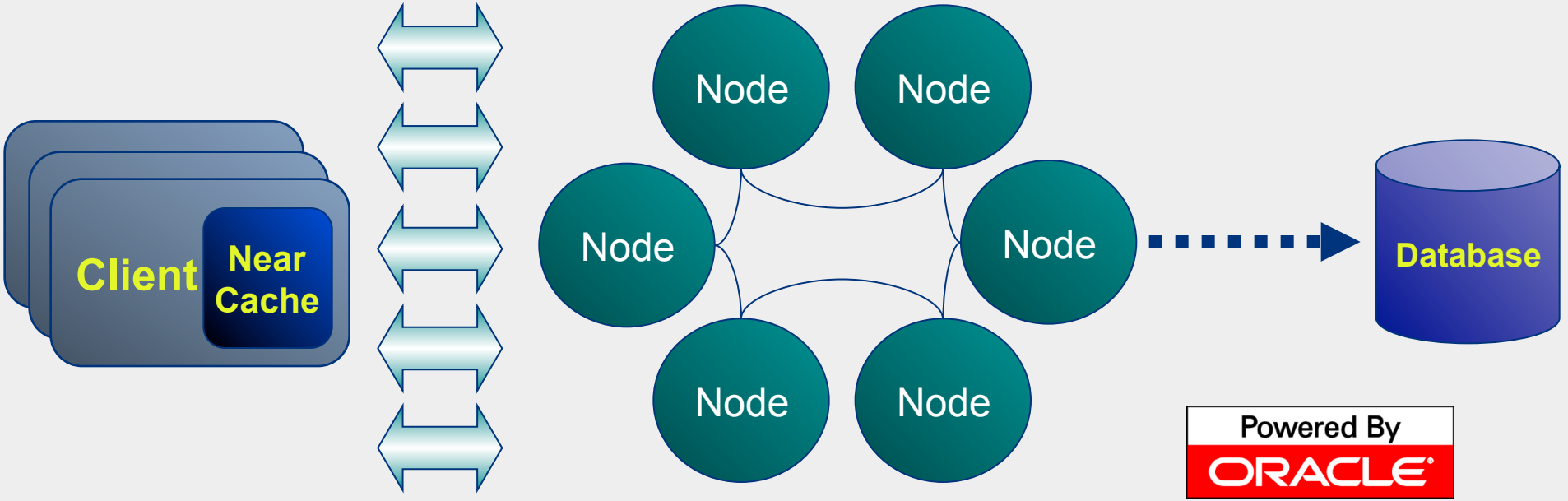


The Post-It Note is a great example of Business Evolution – a product that starts its life in one role, but evolves into something else … otherwise known as Accidental Genius.

Coherence is another good example!!

What is Coherence

A 3rd party distributed data repository with multi-level caching



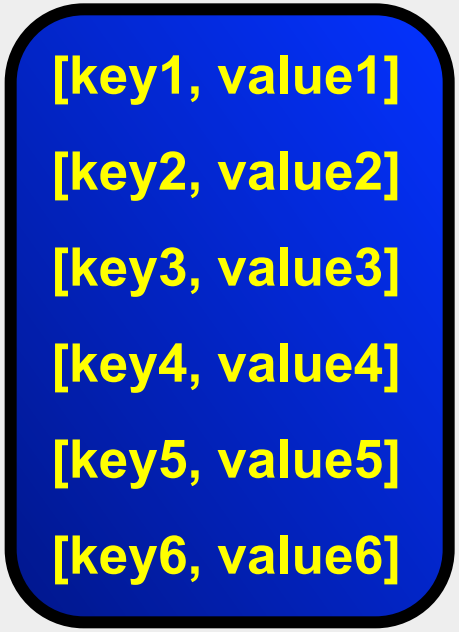

And maybe
something
more.....

RBS
The Royal Bank of Scotland

Putting Coherence in Context

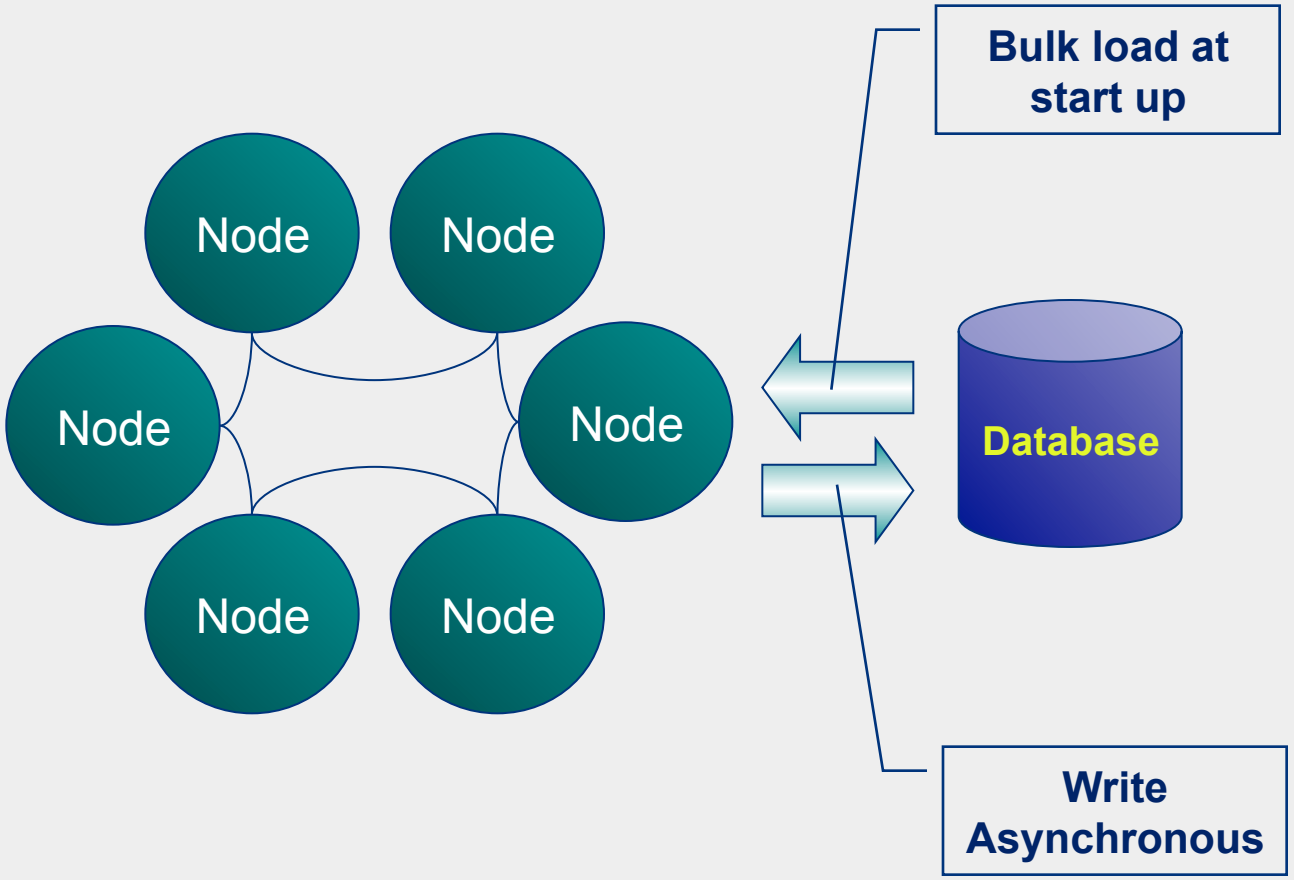
Key Concept: Coherence is just a HashMap

All data is stored as key value pairs



Coherence is not generally used as a Read Through Cache

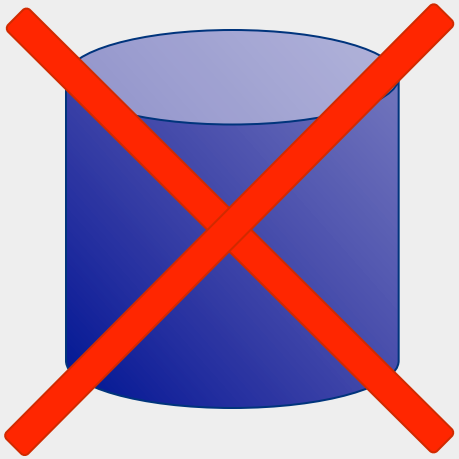
- DB used for persistence only - Coherence is typically prepopulated
- Caching is over two levels (server and client)



Coherence is not a Database

Coherence does not support:

- ACID
- Joins (natively)
- SQL*

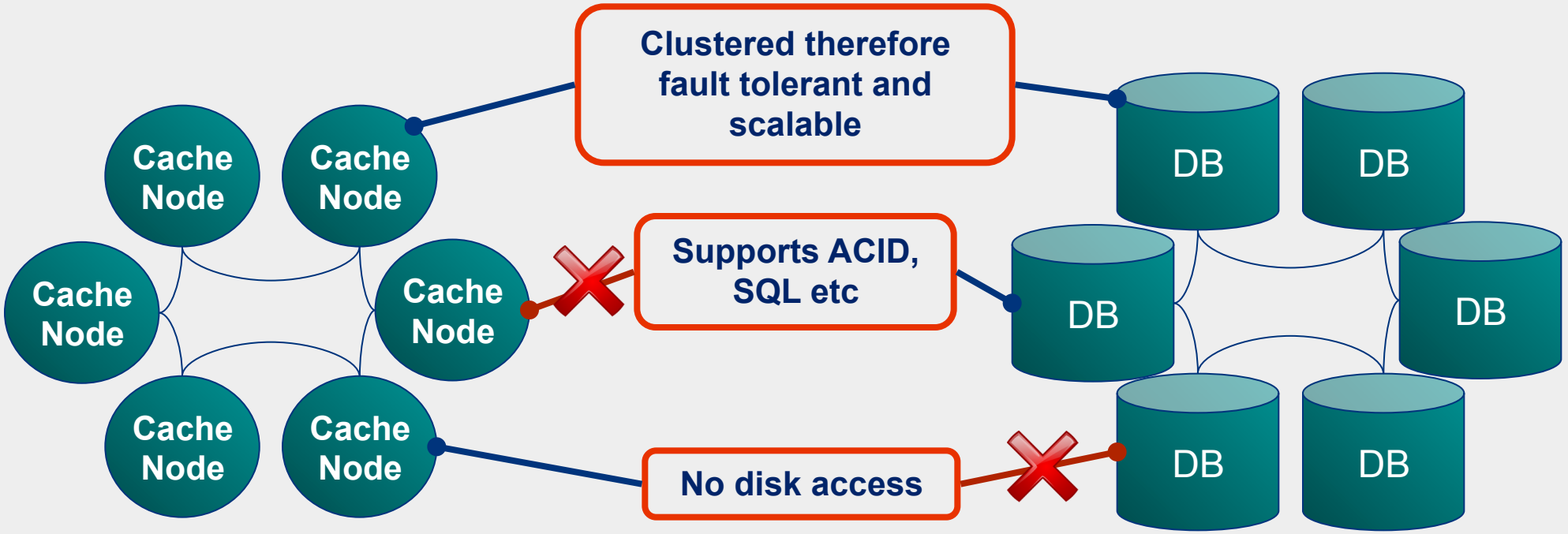


Coherence works to a simpler contract. It is efficient only for simple data access. As such it can do this one job quickly and scalably.

Relationship between RAC and Coherence

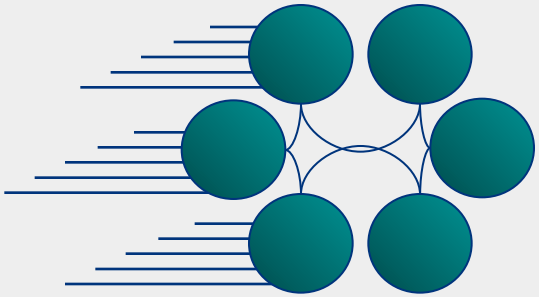
What is RAC?

RAC is a clustered database which runs in parallel over several machines. It supports all the features of vanilla Oracle DB but has better scalability, fault tolerance and bandwidth.

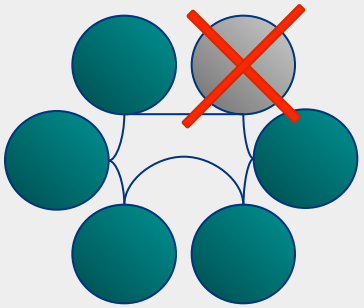


Coherence is:

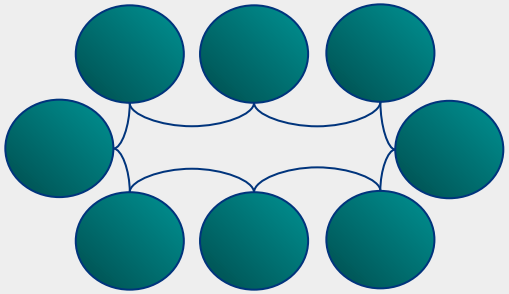
Fast



Fault Tolerant

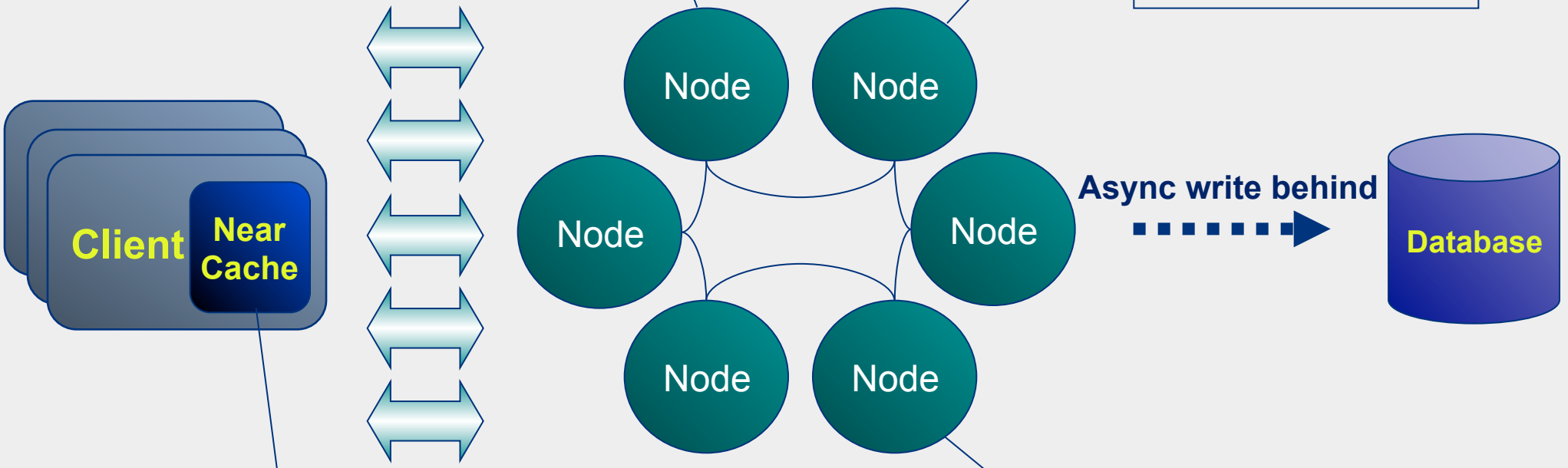


Scalable



Fast

In-memory storage of data – no disk induced latencies (unless you want them).

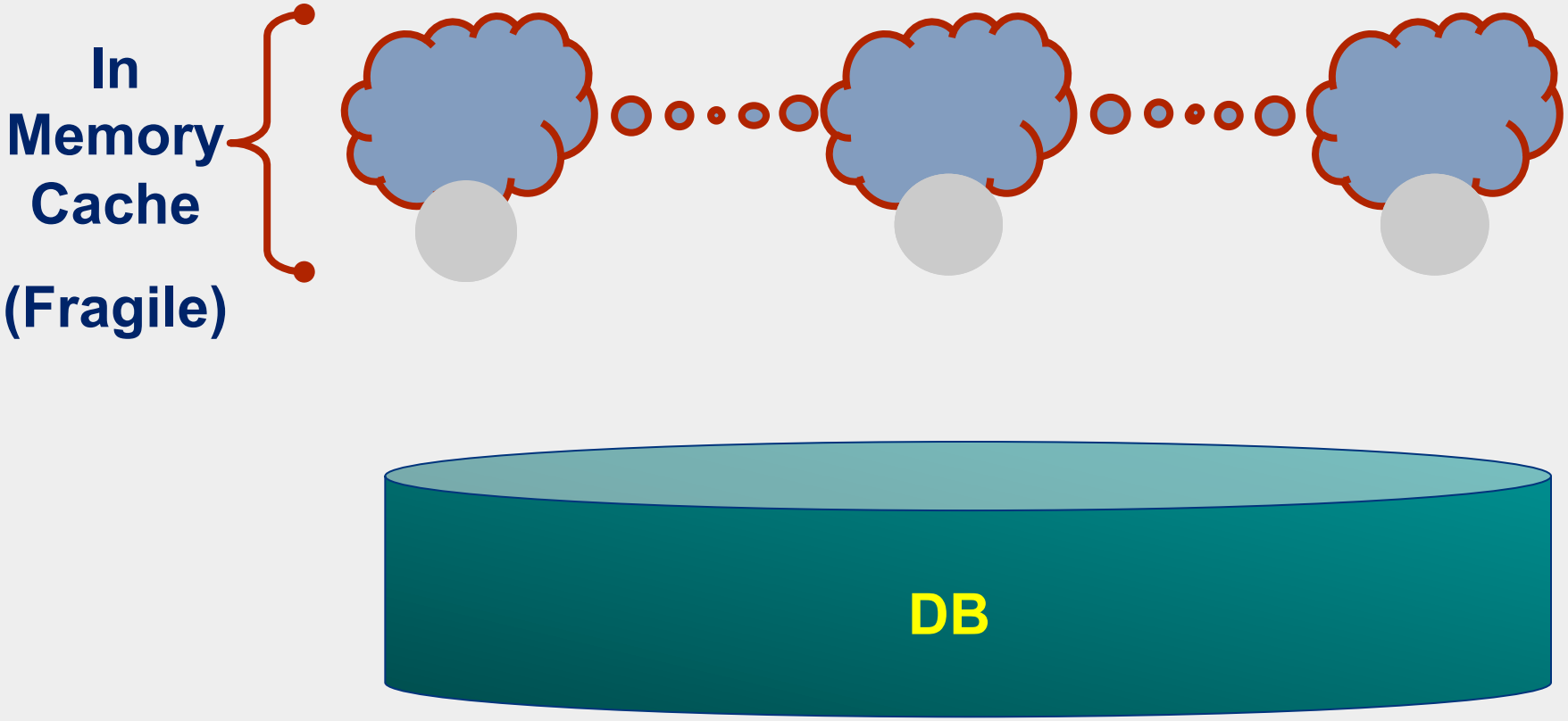


Objects held in serialised form

Near Cache kept Coherent via Proactive update/expiry of data as it changes

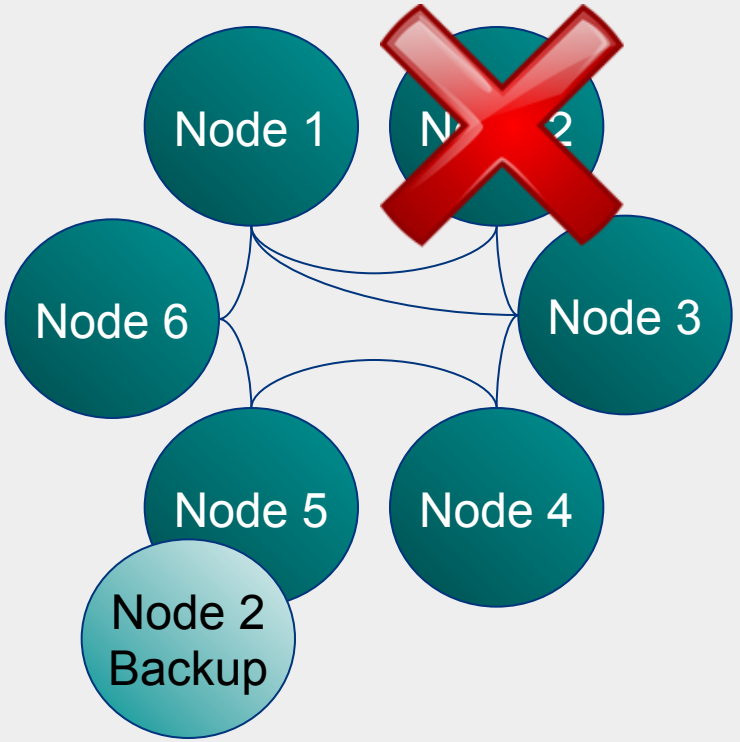
Queries run in parallel where possible (aggregations etc)

Thus a fast caching layer can support much higher load



Fault Tolerance / High Availability

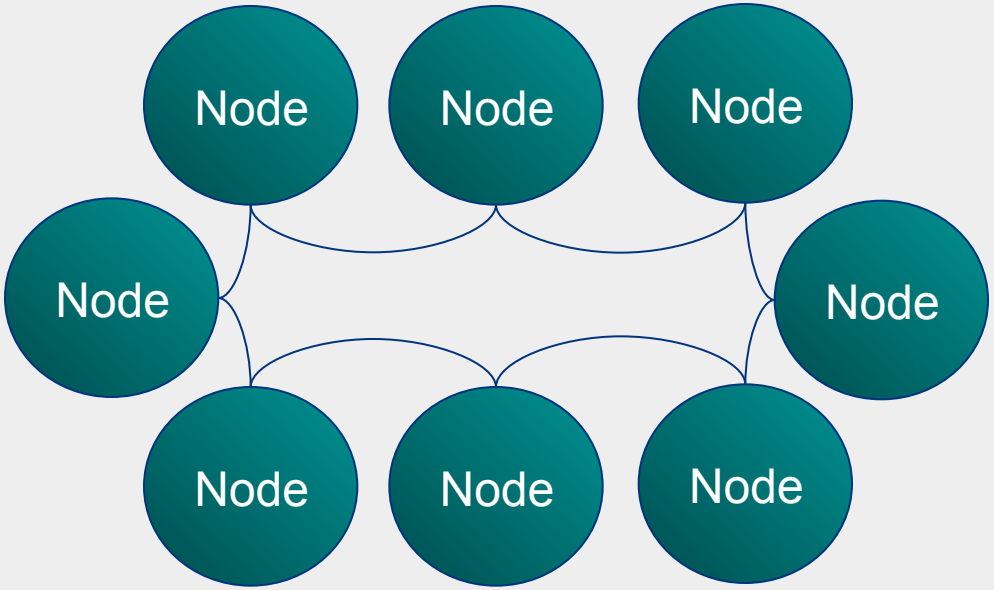
Data is held on at least two machines. Thus, should one fail, the backup copy will still be available.

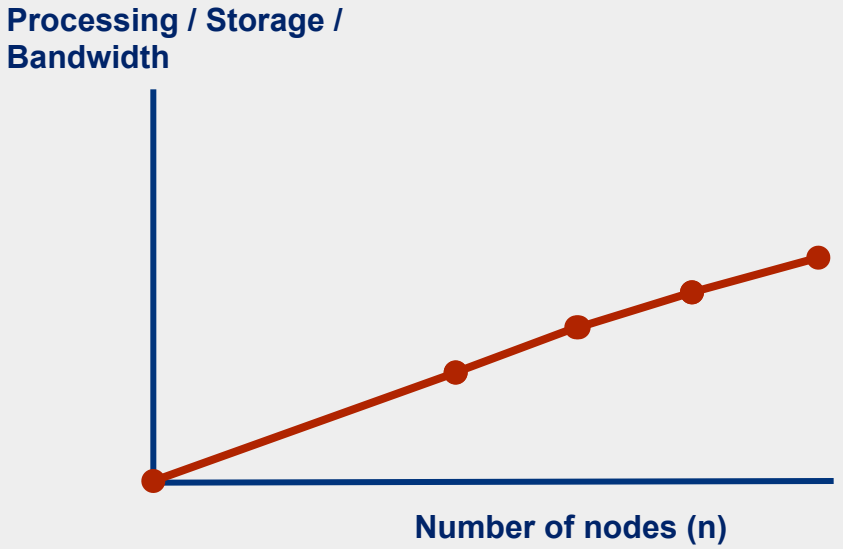


The more machines, the faster failover will be!

Scalable

- Scale the application by adding commodity hardware
- Coherence automatically detects new cluster members
- Near-linear scalability due to ***partitioned*** data





So how resilient is this architecture?

Not that resilient:

- Single machine failure will be tolerated.
- Concurrent machine failure will cause data loss.

Key Point: Resilience is sacrificed for speed

Lets reiterate the difference between Coherence and a database.

- Coherence works to a simpler contract. It is efficient only for simple data access. As such it can do this one job quickly and scalably.
- Databases are constrained by the wealth of features they must implement. Most notably (from a latency perspective) ACID.
- In the bank we are often happy to sacrifice ACID etc for speed and scalability.

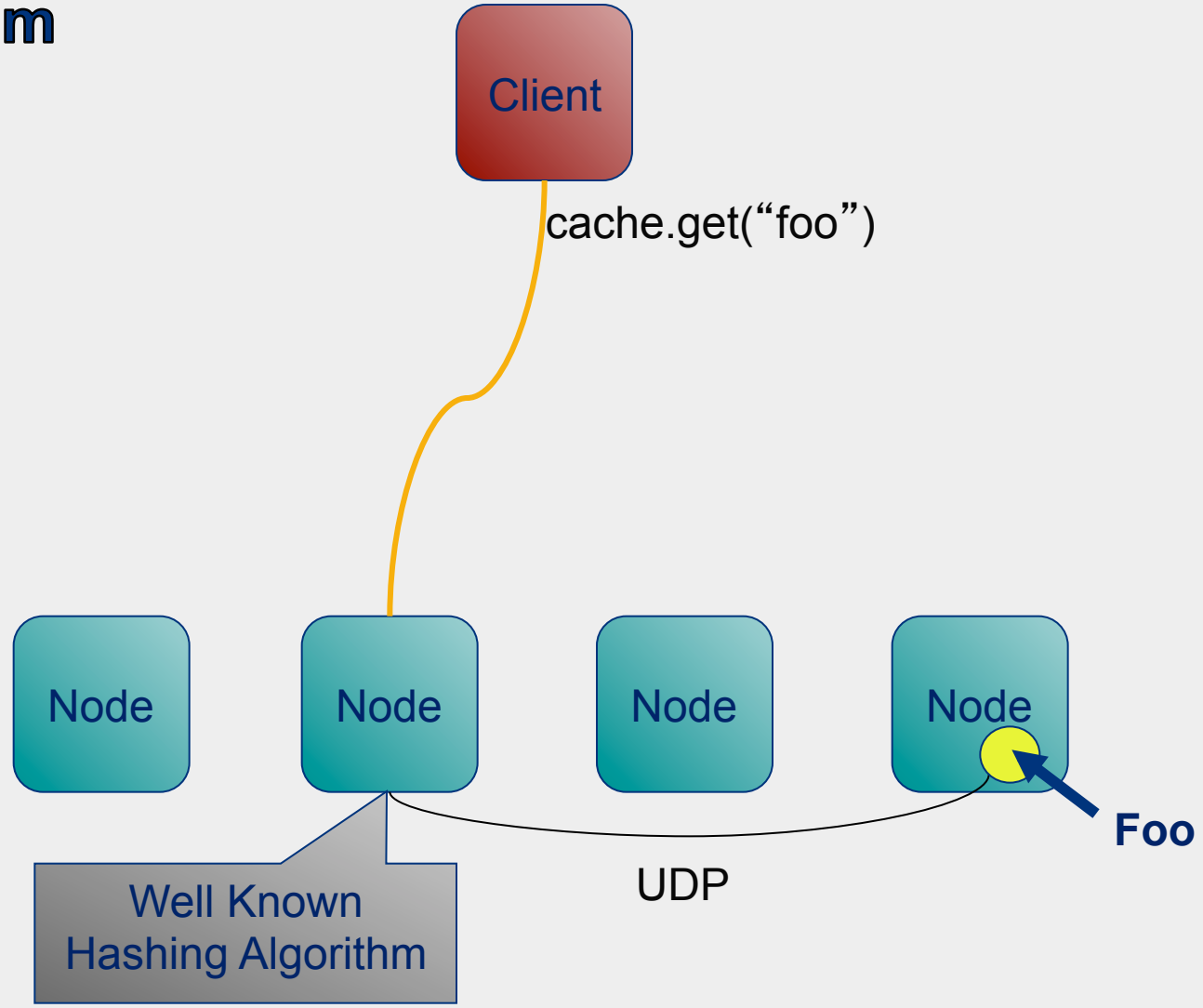
Summary so far…

Summary so far…

Summary so far…

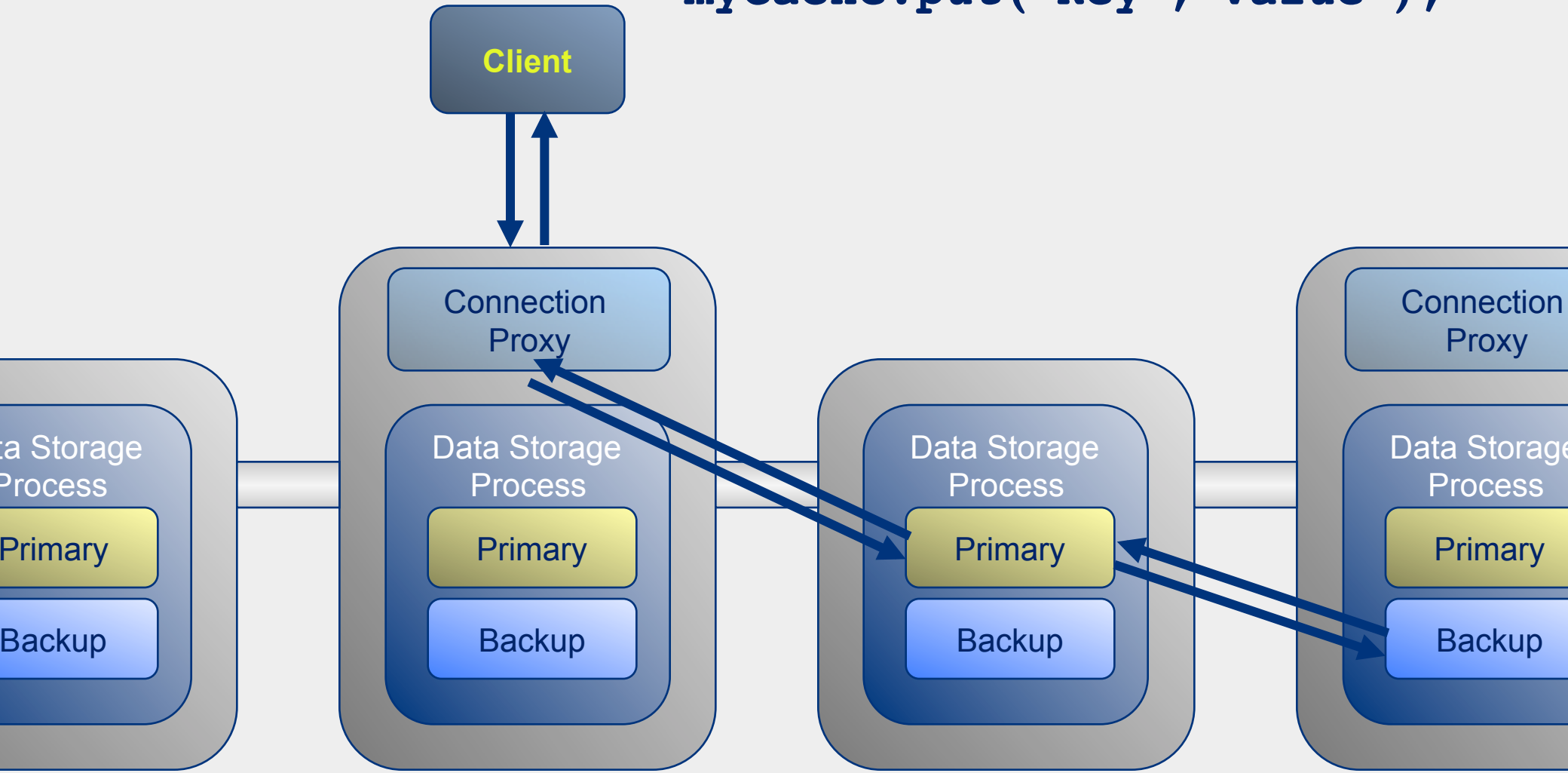
Now lets delve a little deeper…

Communication Between Nodes: Well Known Hashing Algorithm



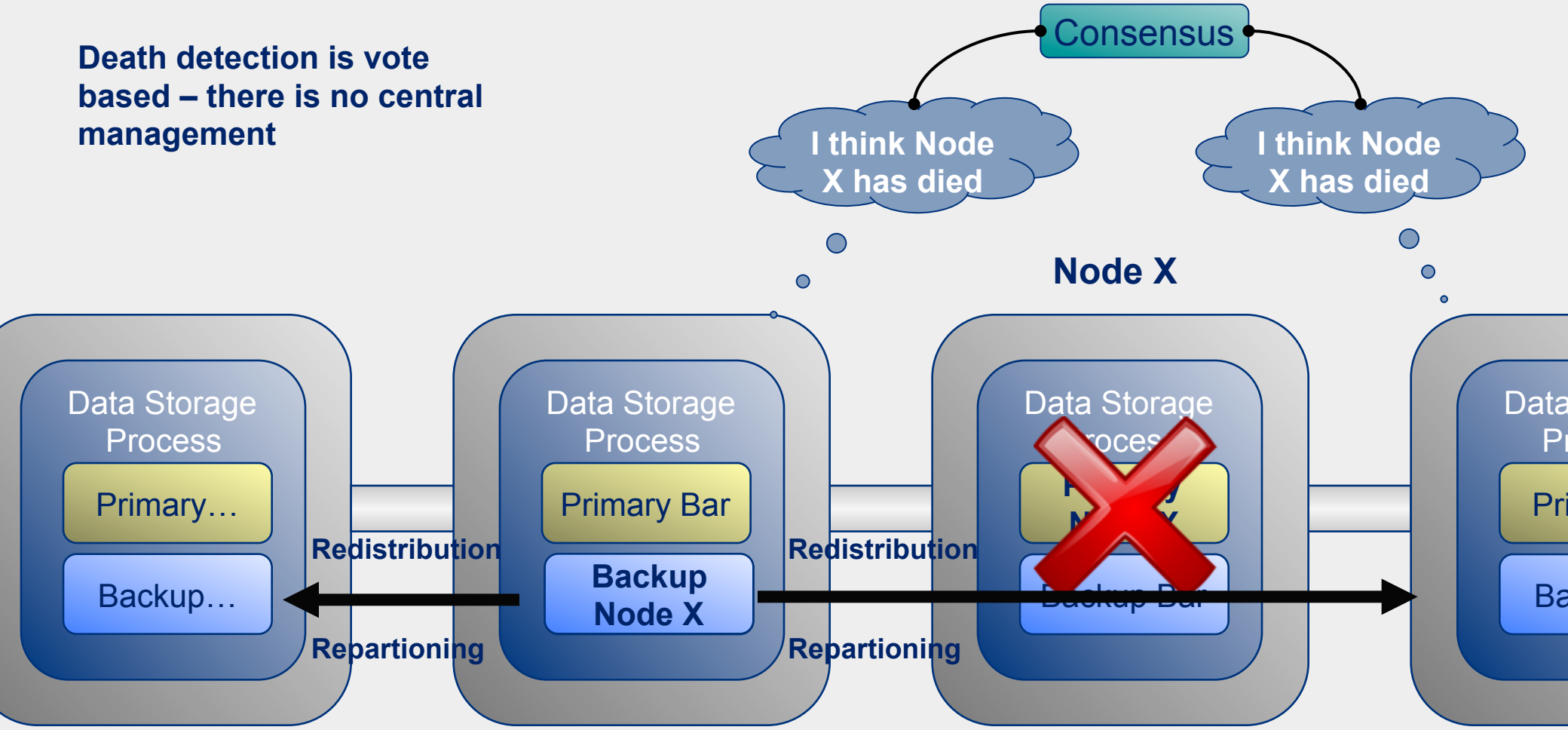
Writing Data to the Cluster

```
myCache.put("Key","Value");
```

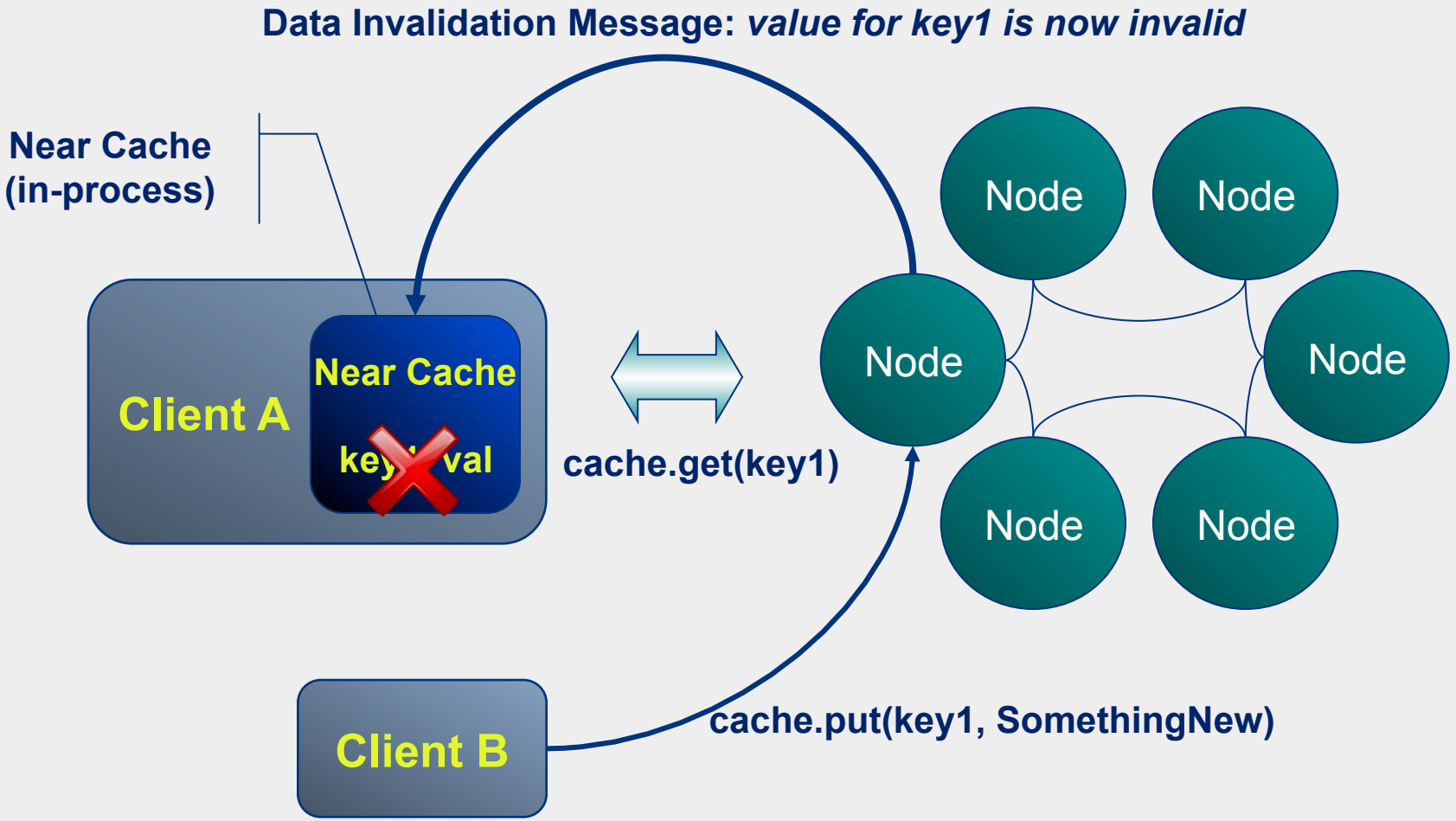


Node Failure

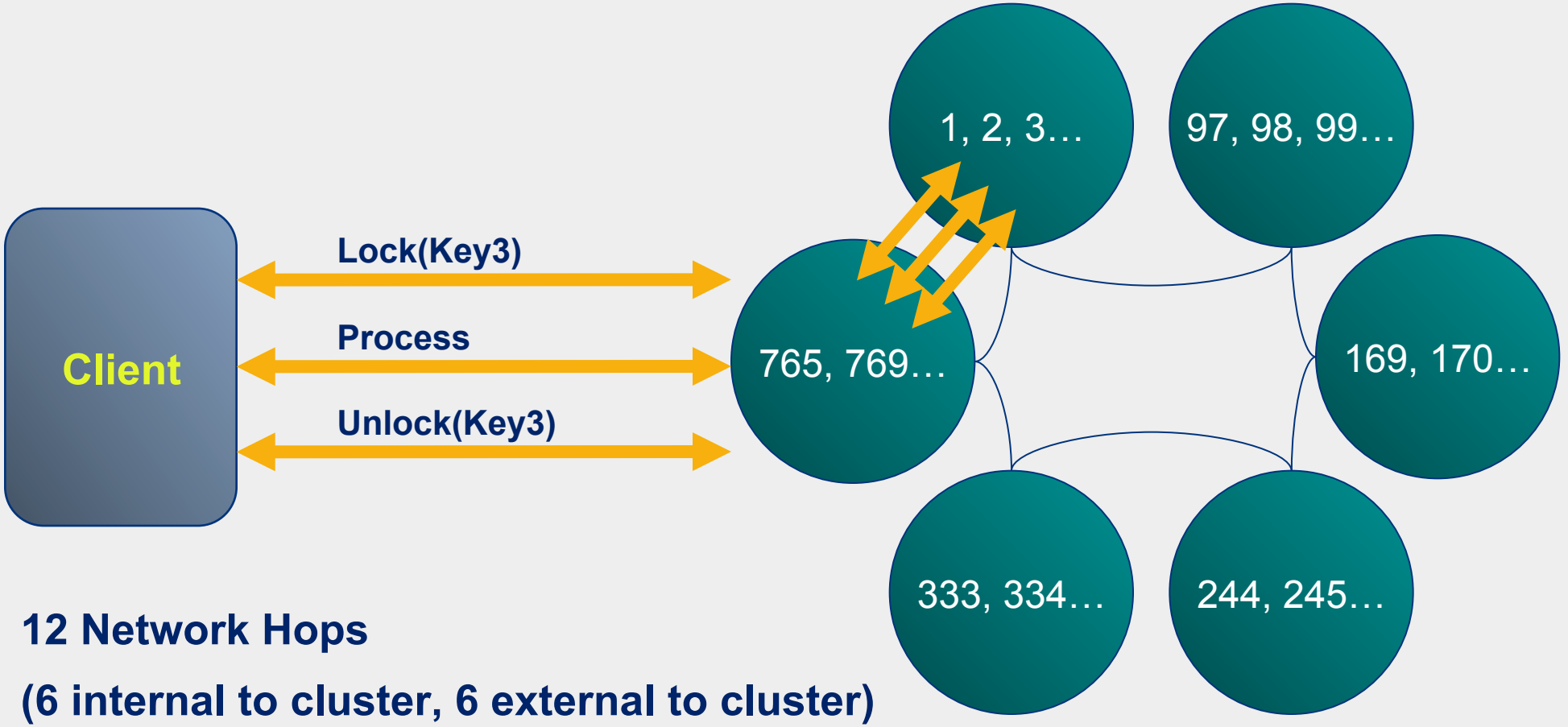
Death detection is vote based – there is no central management



Near Caching - Just in Time provisioning of locally cached data



Locking and Synchronous Operations

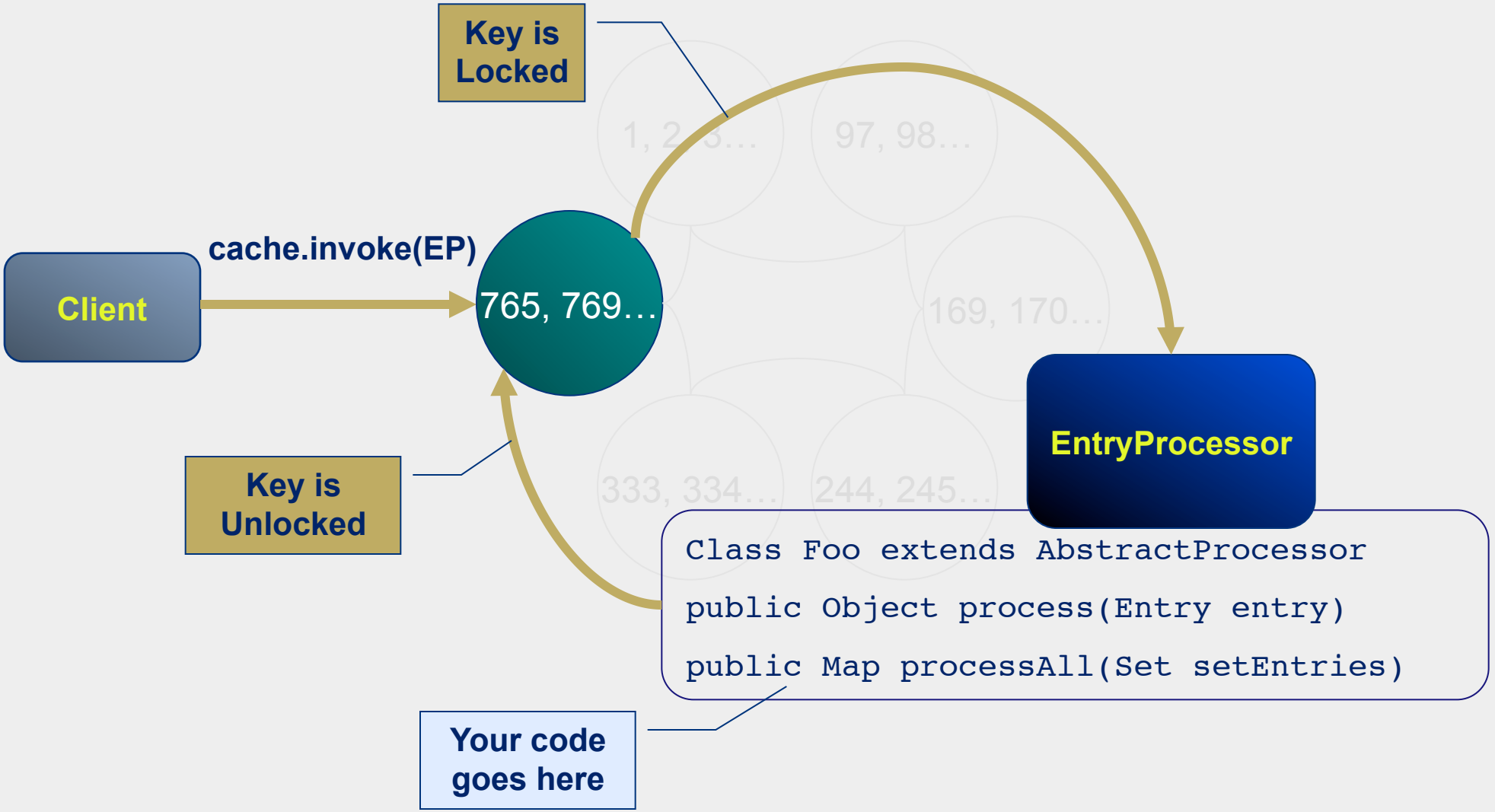


12 Network Hops

(6 internal to cluster, 6 external to cluster)

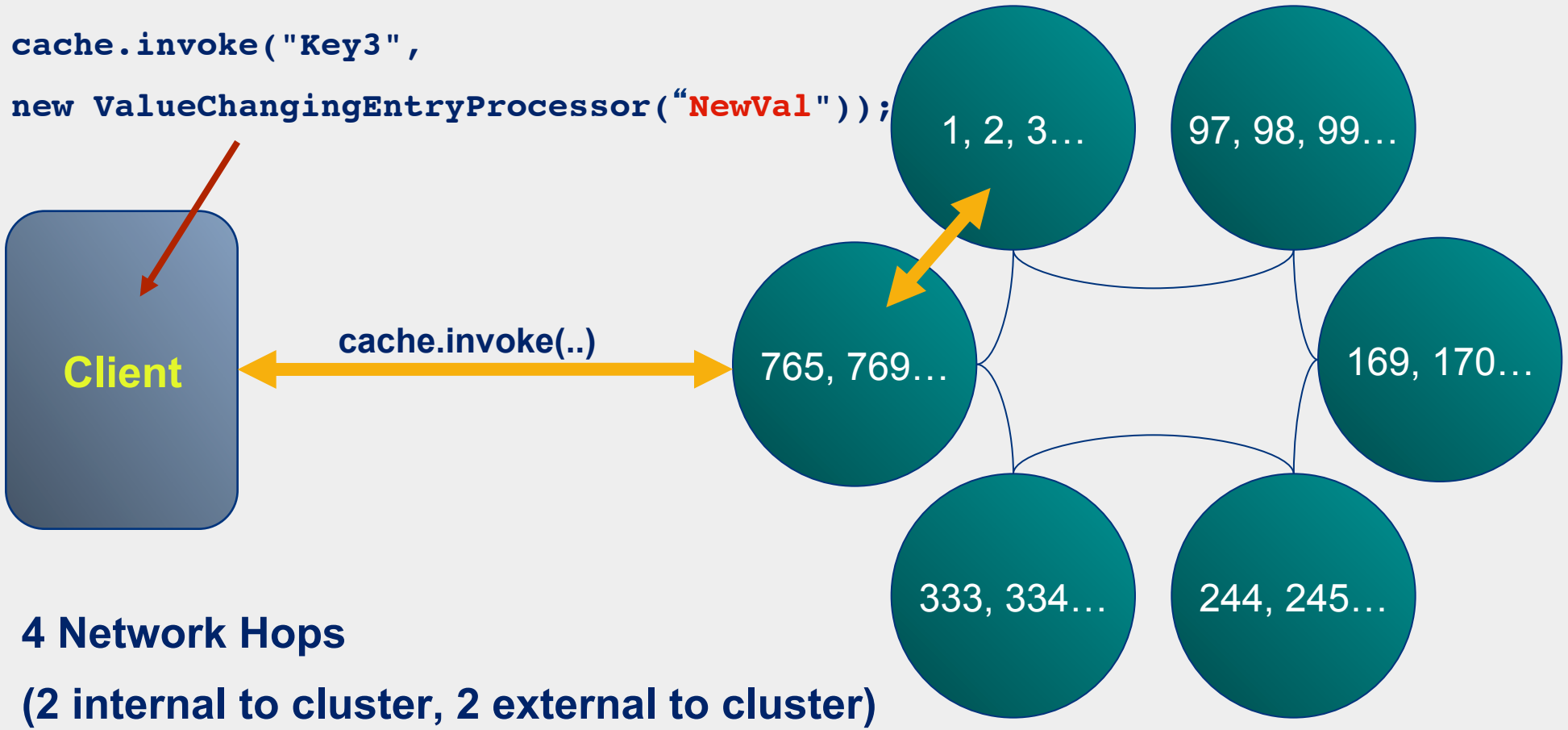
Entry Processors

Analogous to: Stored Procedures



Entry Processors

```
cache.invoke("Key3",
new ValueChangingEntryProcessor("NewVal"));
```



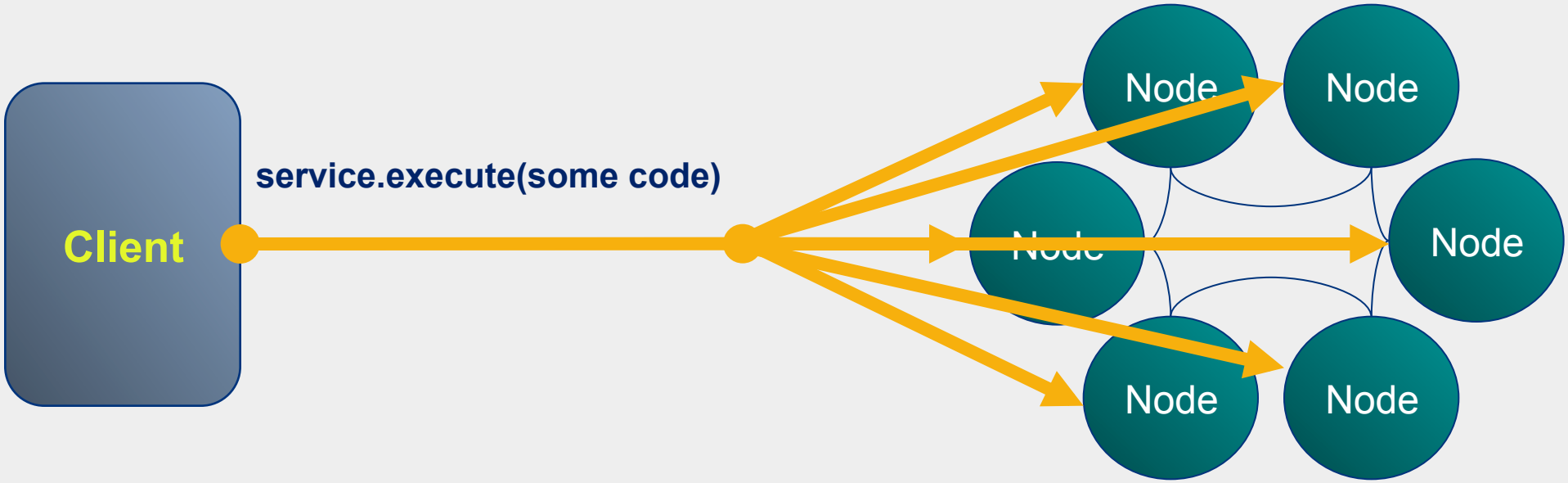
4 Network Hops

(2 internal to cluster, 2 external to cluster)

Invocables

Analogous to: Grid Task

```
service.execute(new GCAgent(), null, null);
```

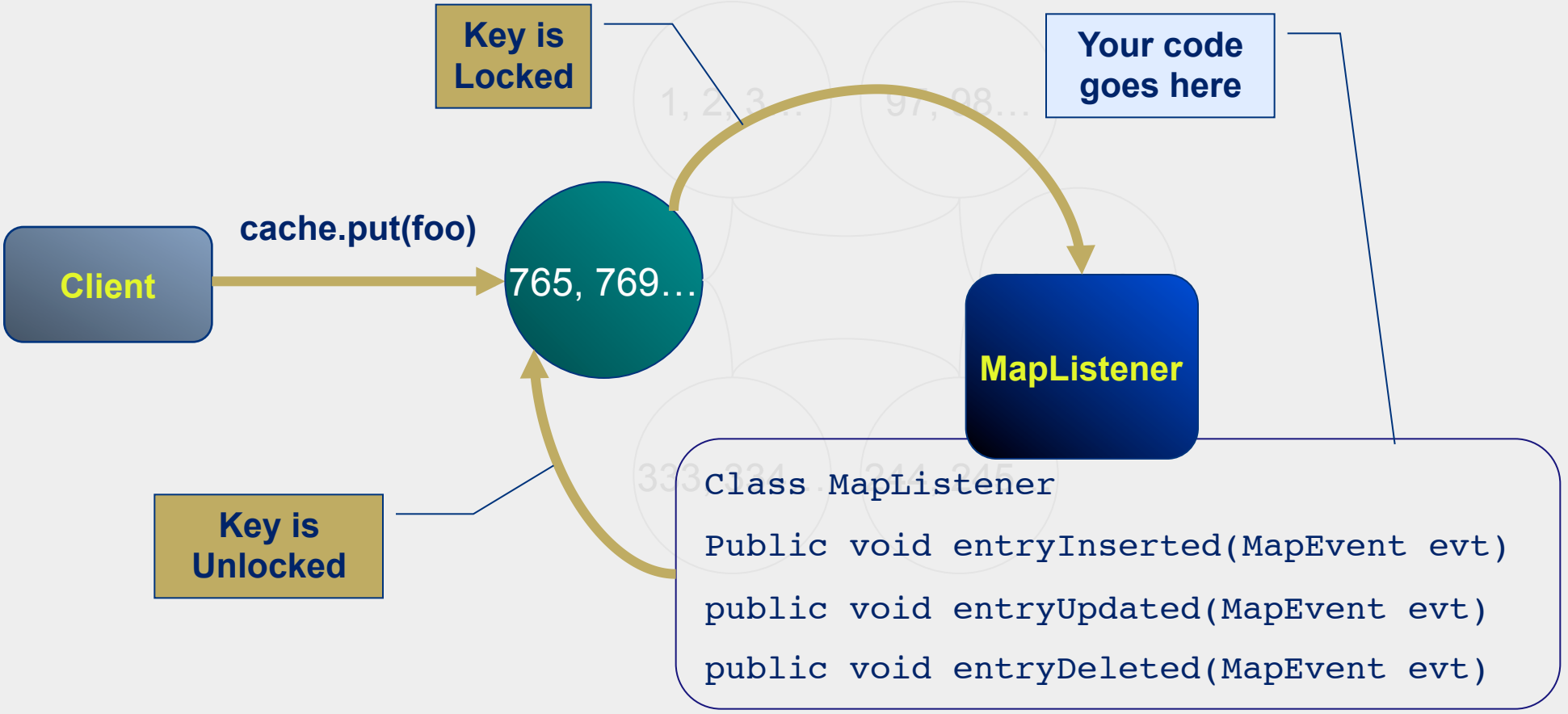


Run any arbitrary piece of code on any or all of the nodes

Backing Map Listeners / Triggers

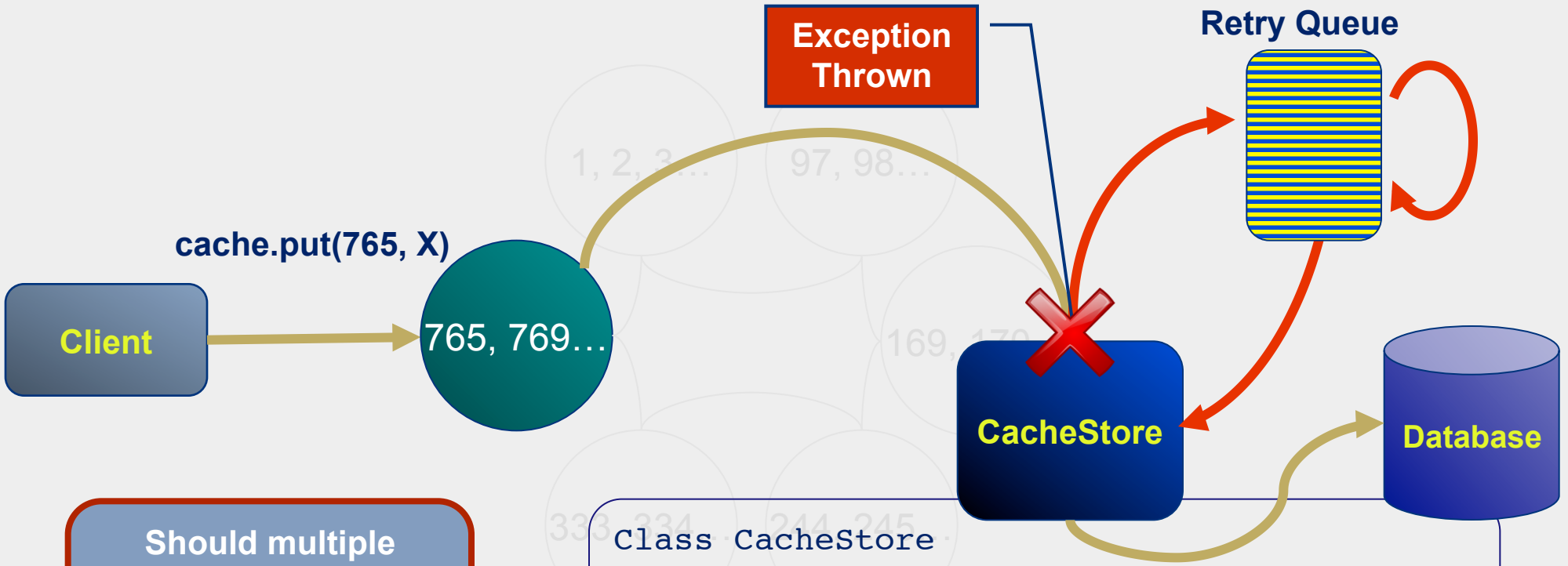
Analogous to: Triggers

Backing Map Listeners / Triggers allow code to be run in response to a cache event such as an entry being added, updated or deleted.

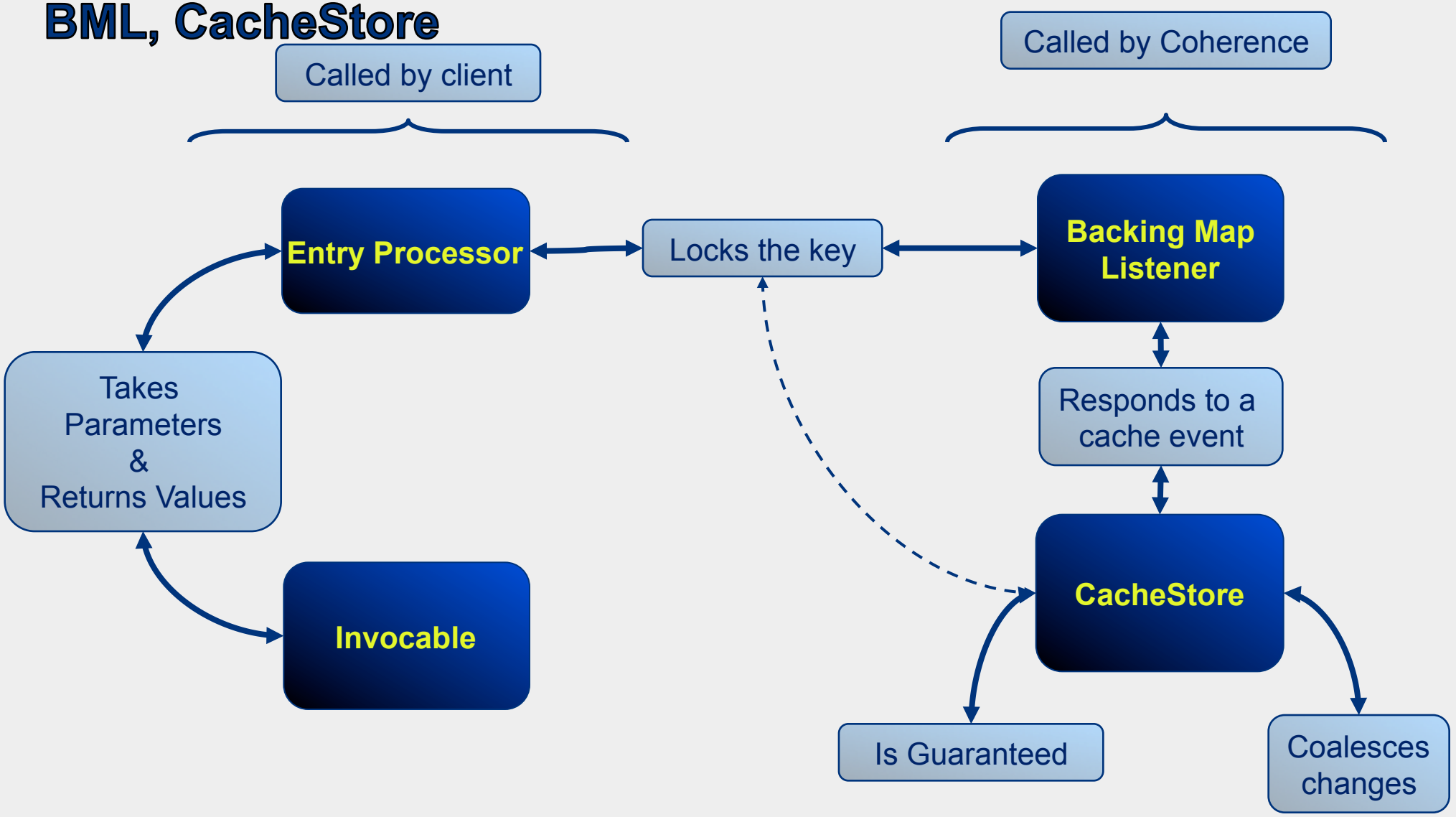


Cache Stores

Analogous to: Triggers (but with fault tolerance and built in retry)



Comparing Entry Processor, Invocable, BML, CacheStore

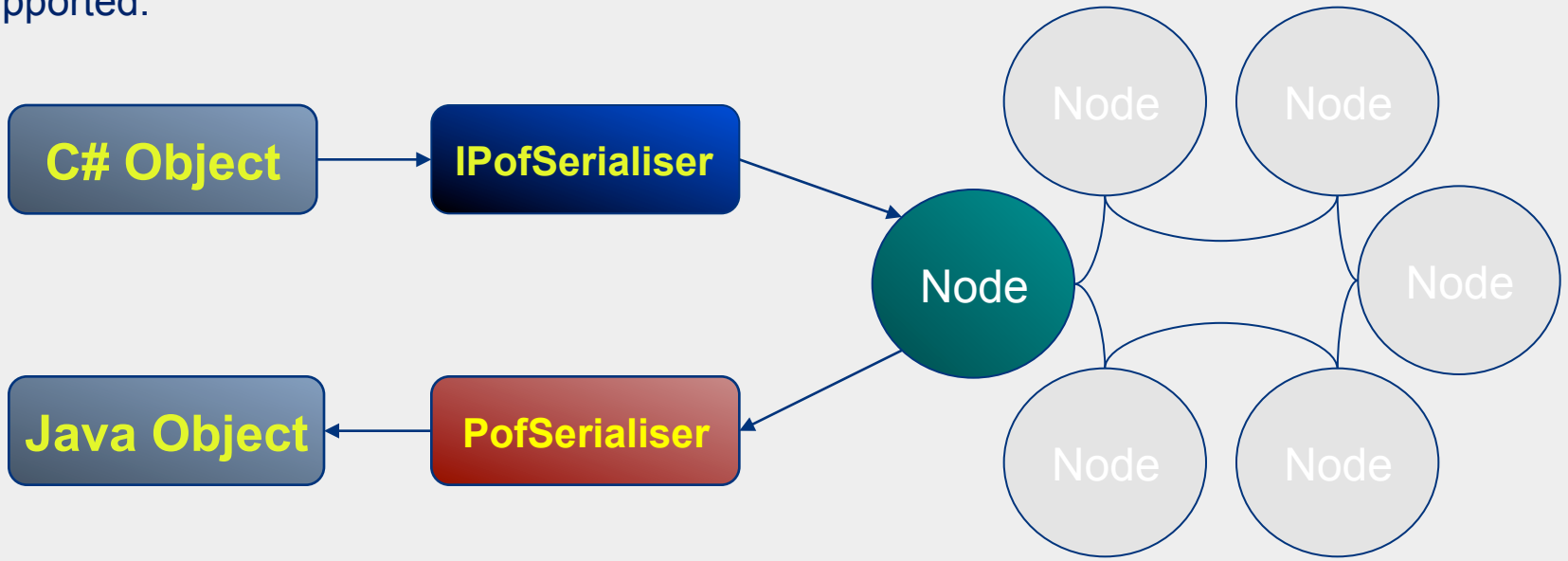


Technologies Serviced

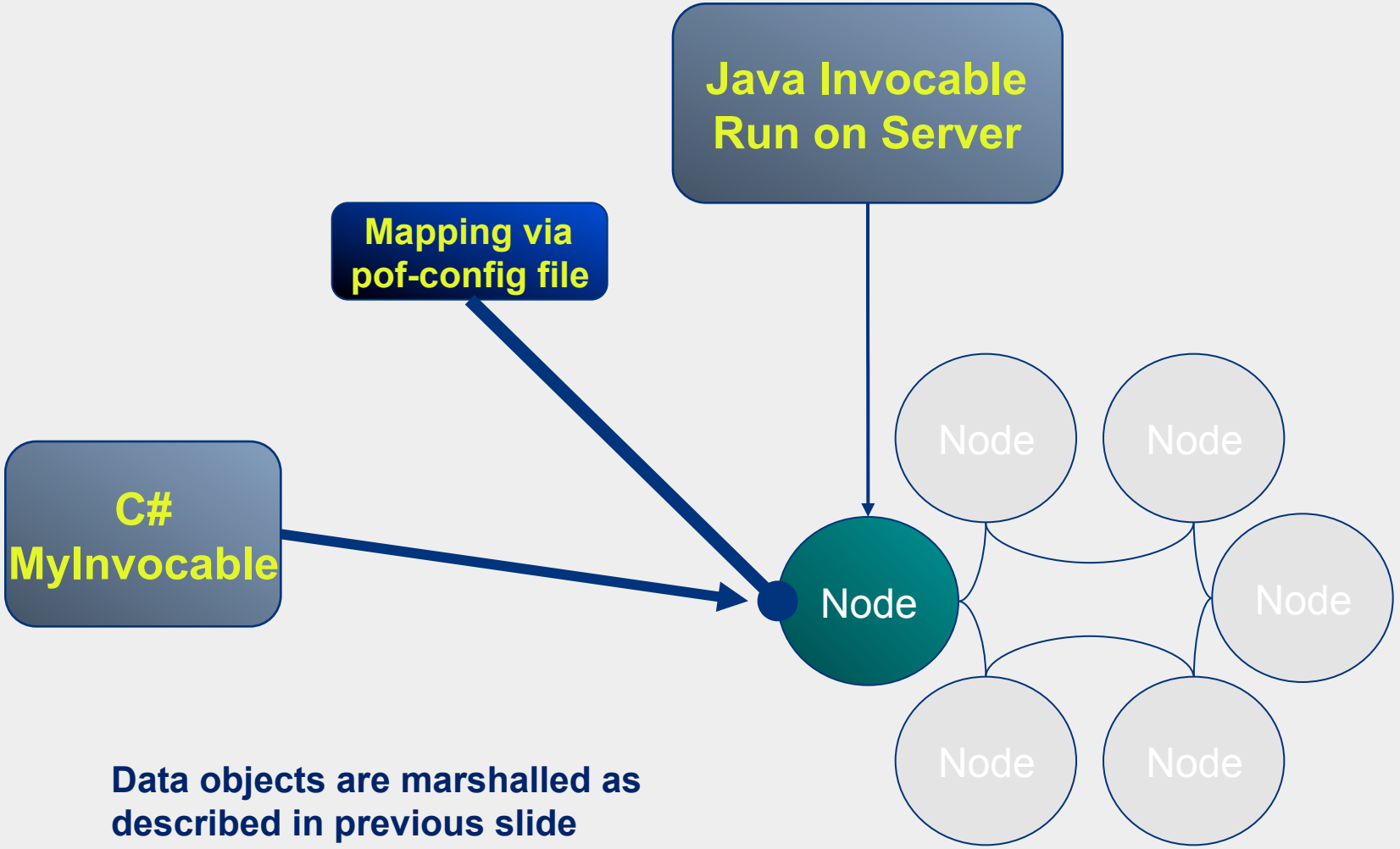
All cluster side programming must be done in Java. However clients can be:

- Java
- C#
- C++

Serialisation is done to an intermediary binary format known as POF. This allows theoretical transformation from POF directly to any language. Currently only Java, C# and C++ are supported.



Calling Server Side Processing Code from C#

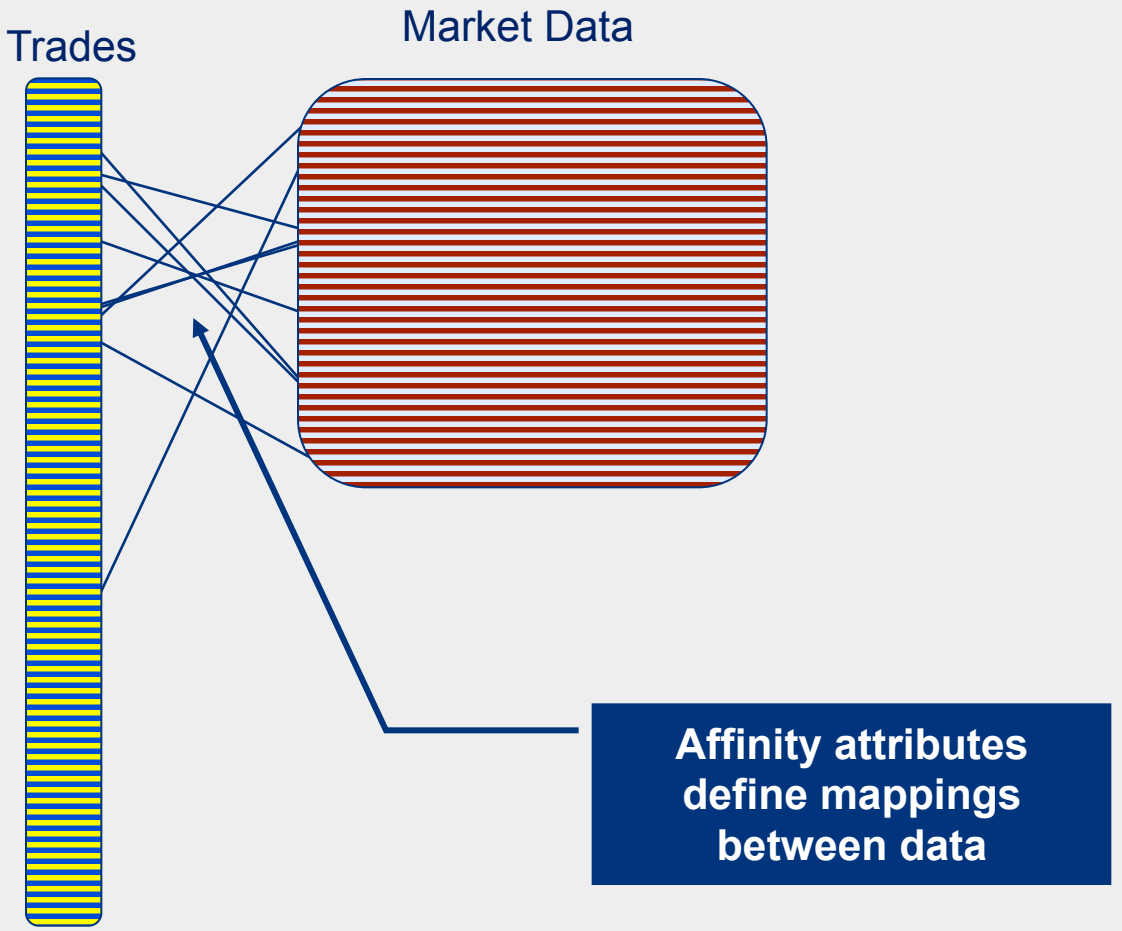


Data objects are marshalled as described in previous slide

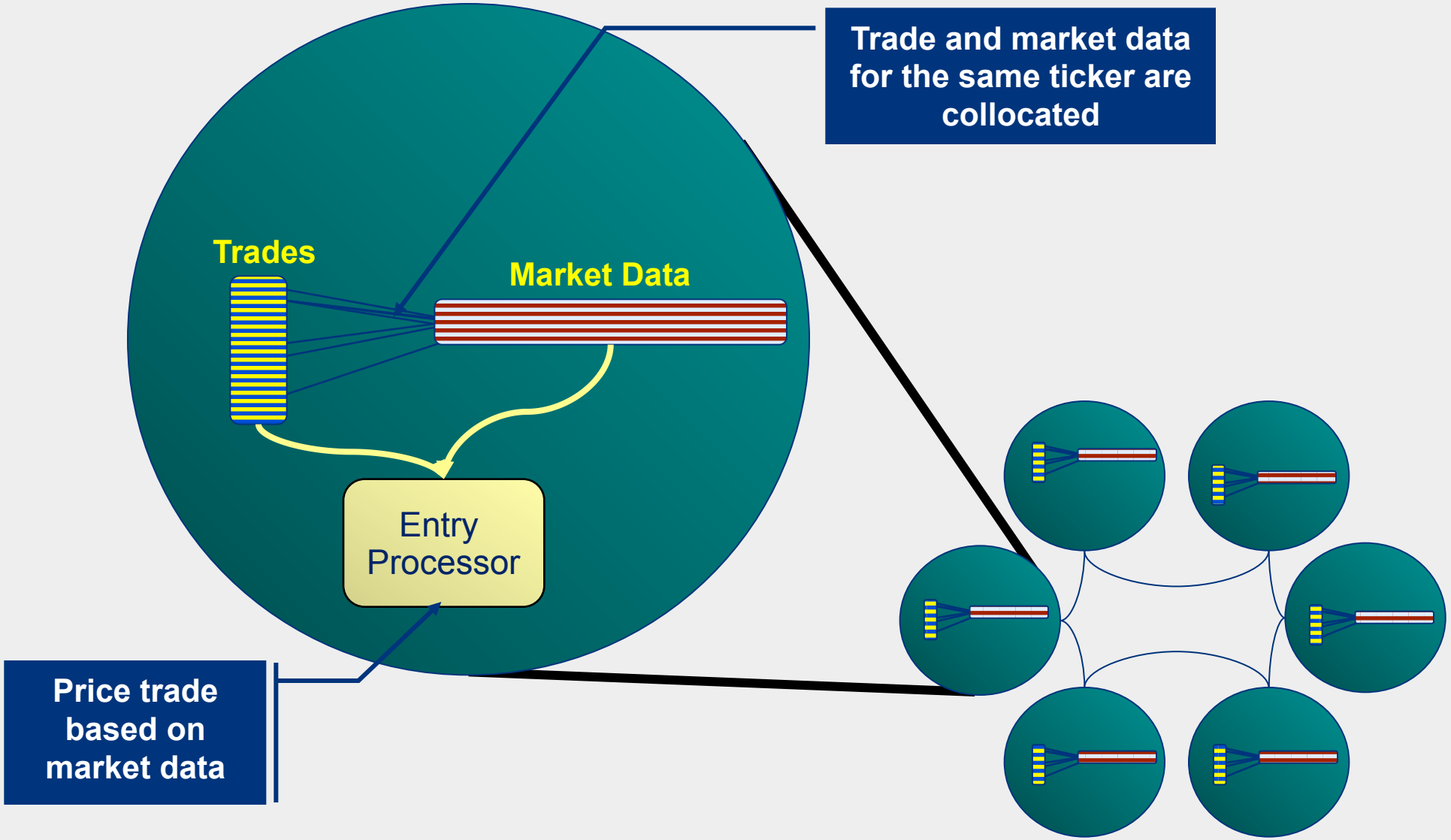
Running Processing in the Cache

Data Affinity

Set up associations between **affinity** **attributes** so that they are stored on the same machine



Associated entries are held together on each machine



This is key requirement is to run processing inside the cache without incurring the penalties associated with remote data access.

Wire Efficiency - An alternative model: Send the processing to the data

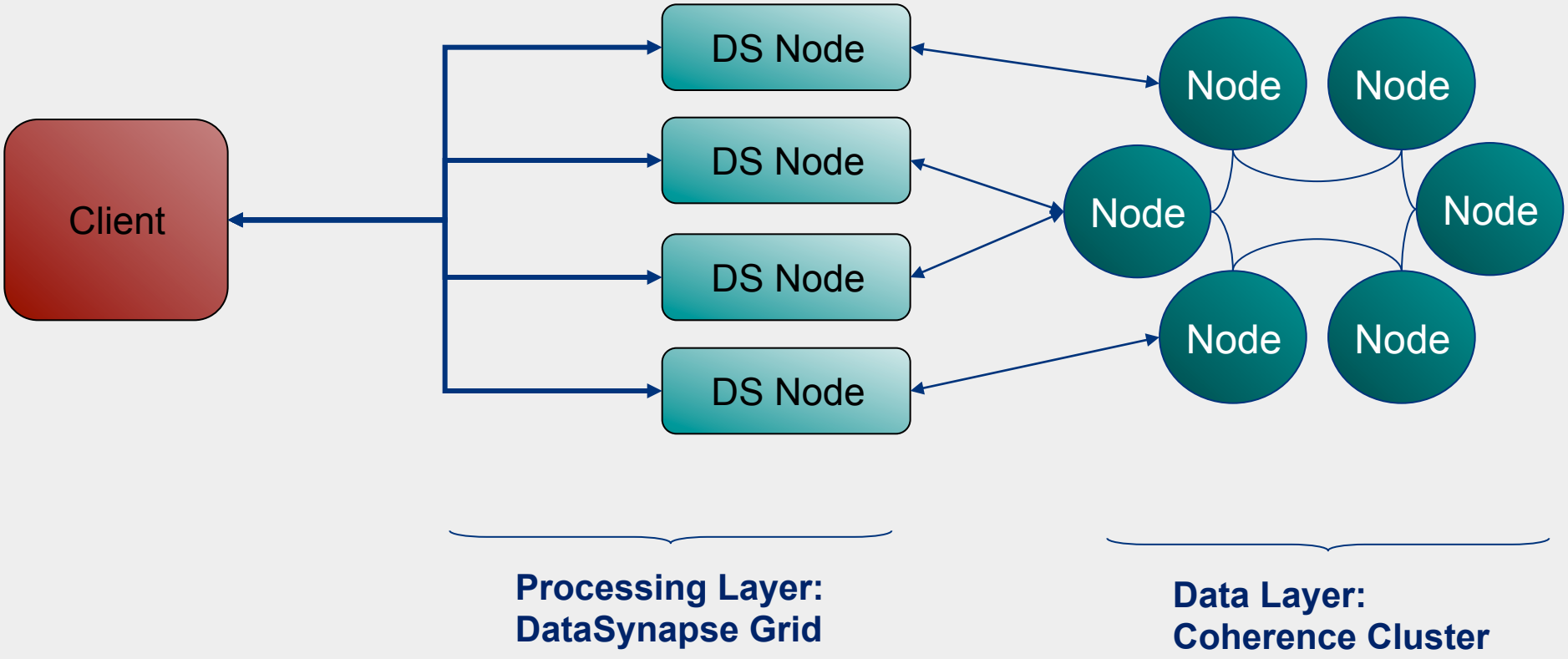
Send data to code

```
Public class foo{
      Public void bar(){
            //do some stuff
      }
}
```

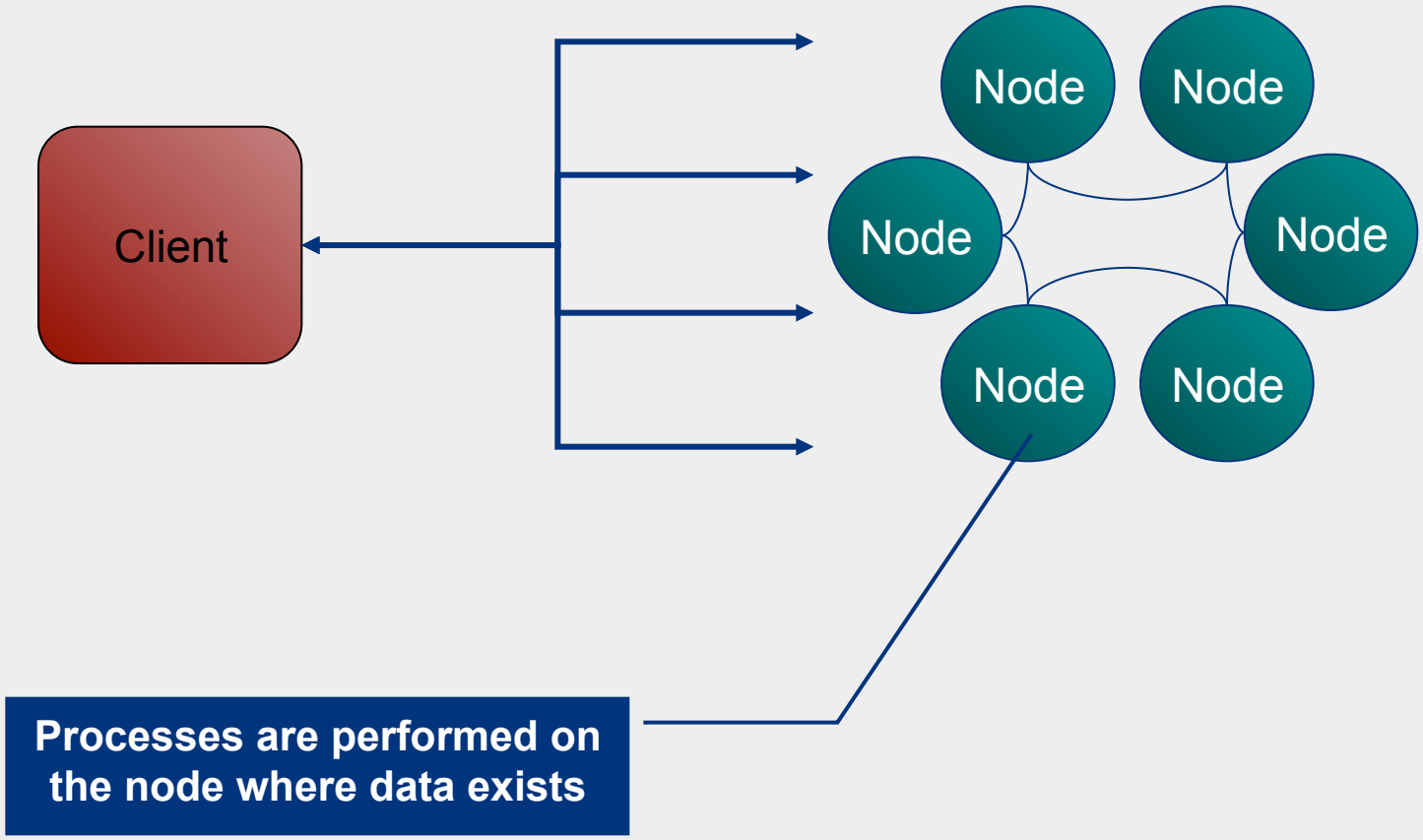
```
Public class foo{
      Public void bar(){
            //do some stuff
      }
}
```

Send code to data

Merging Data and Processing

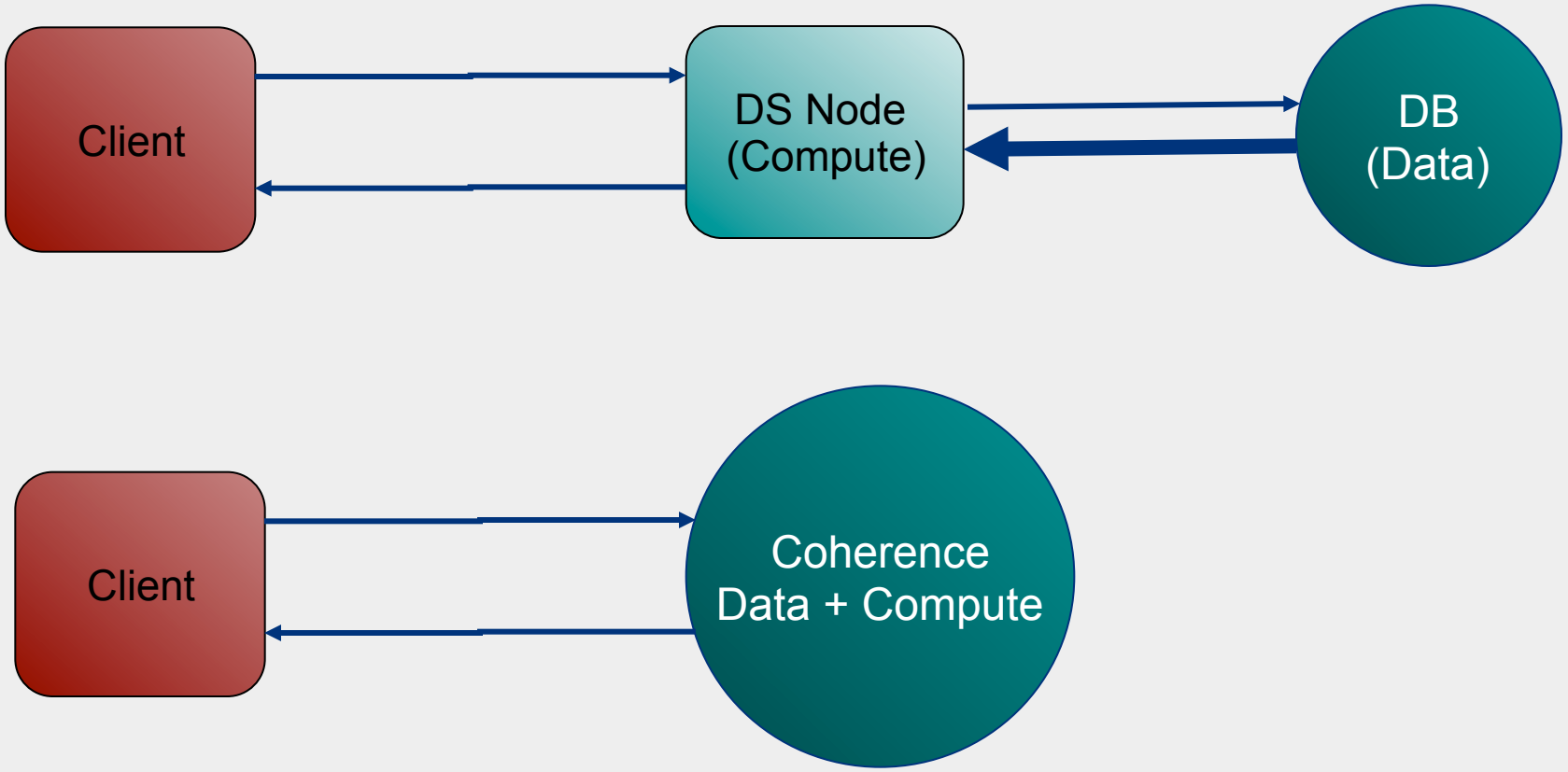


… becomes

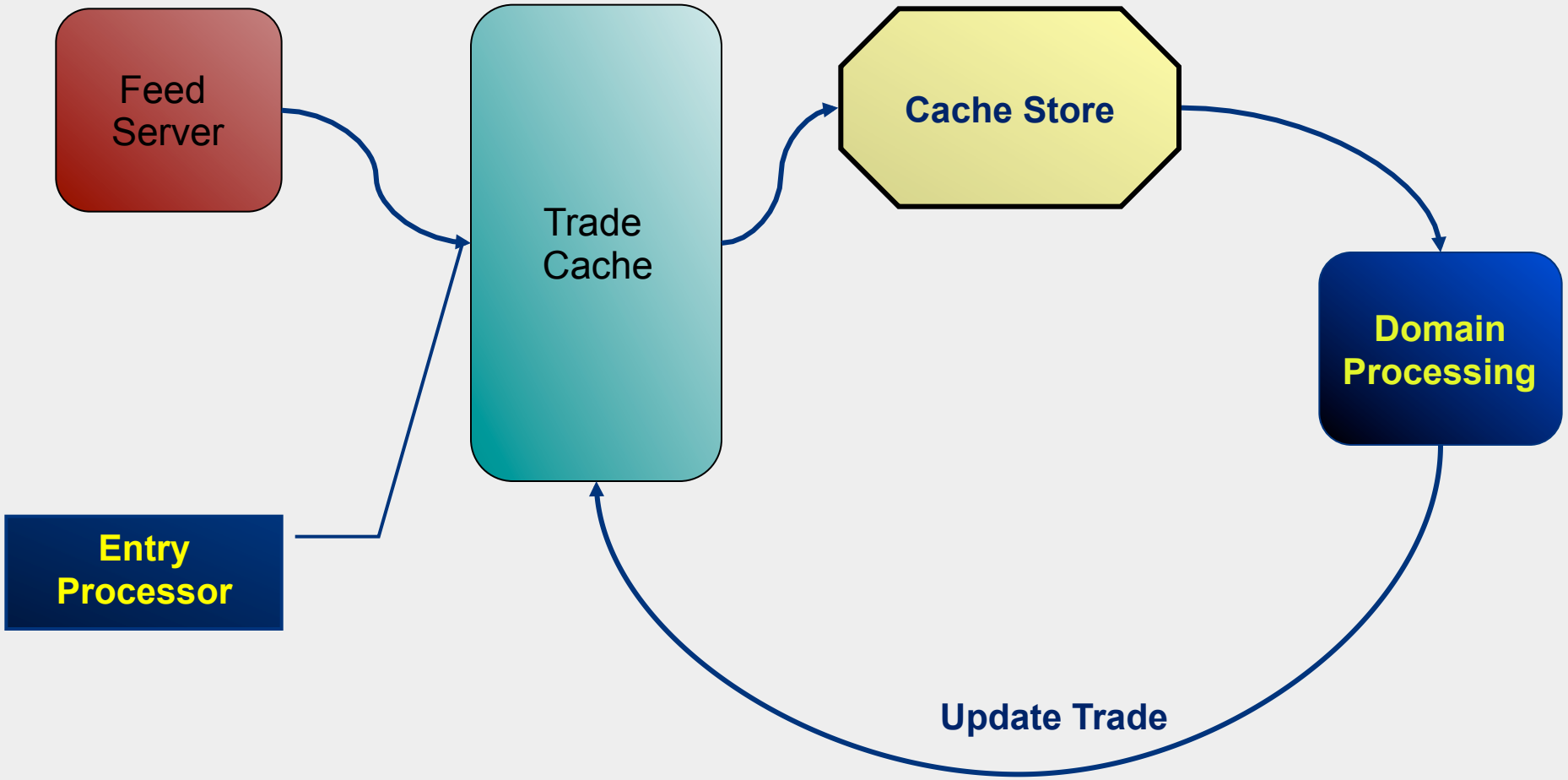

Client
Node
Node
Node
Node
Node
Node
Processes are performed on the node where data exists

We decrease transactional latency by executing processing where the data resides

For example compare data flows in Grid vs. Coherence based computations

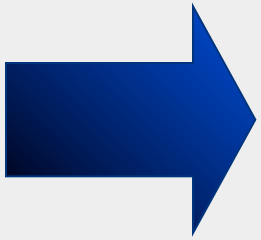


This leads to the concept of an Application-Centric Deployment



Coherence as an Application Container

- Free distribution of processing across multiple machines
- Free fault tolerance
- Free scalability to potentially thousands of machines



A very enticing proposition for new applications

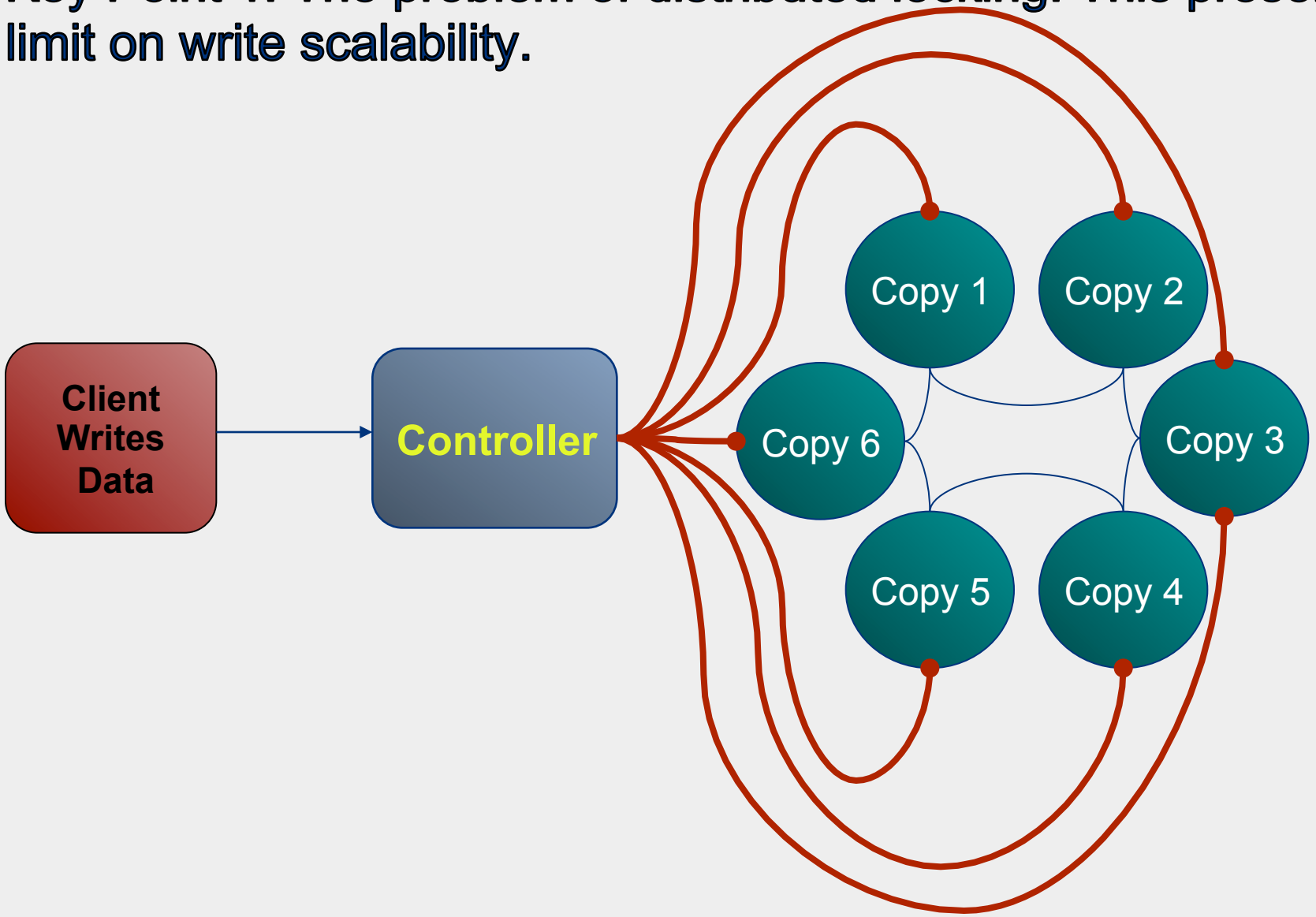
So if Coherence is so great why don't we do it all the time? Why use the Compute Grid at all?

Coherence is not suitable for large scale processor intensive tasks (think Monte Carlo simulations etc). Why?

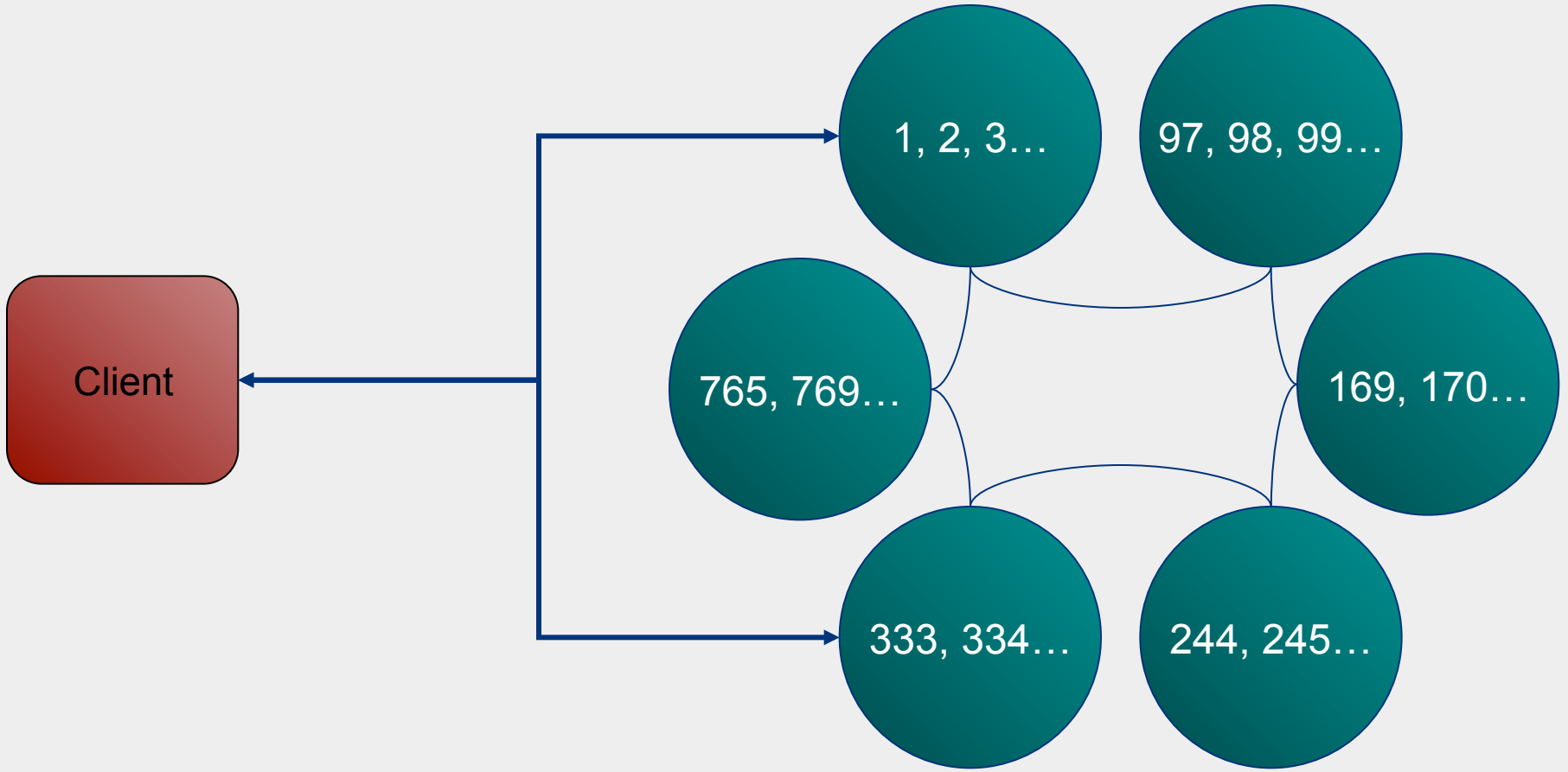
- **Compute grids provide much more control of the execution environment (priorities, a UI etc)**
- **The grid is far more scalable** in terms of compute power (dynamic provisioning of engines etc).
- **The grid is much cheaper** (per core) than Coherence.

So those key points again…

Key Point 1: The problem of distributed locking. This presents a limit on write scalability.

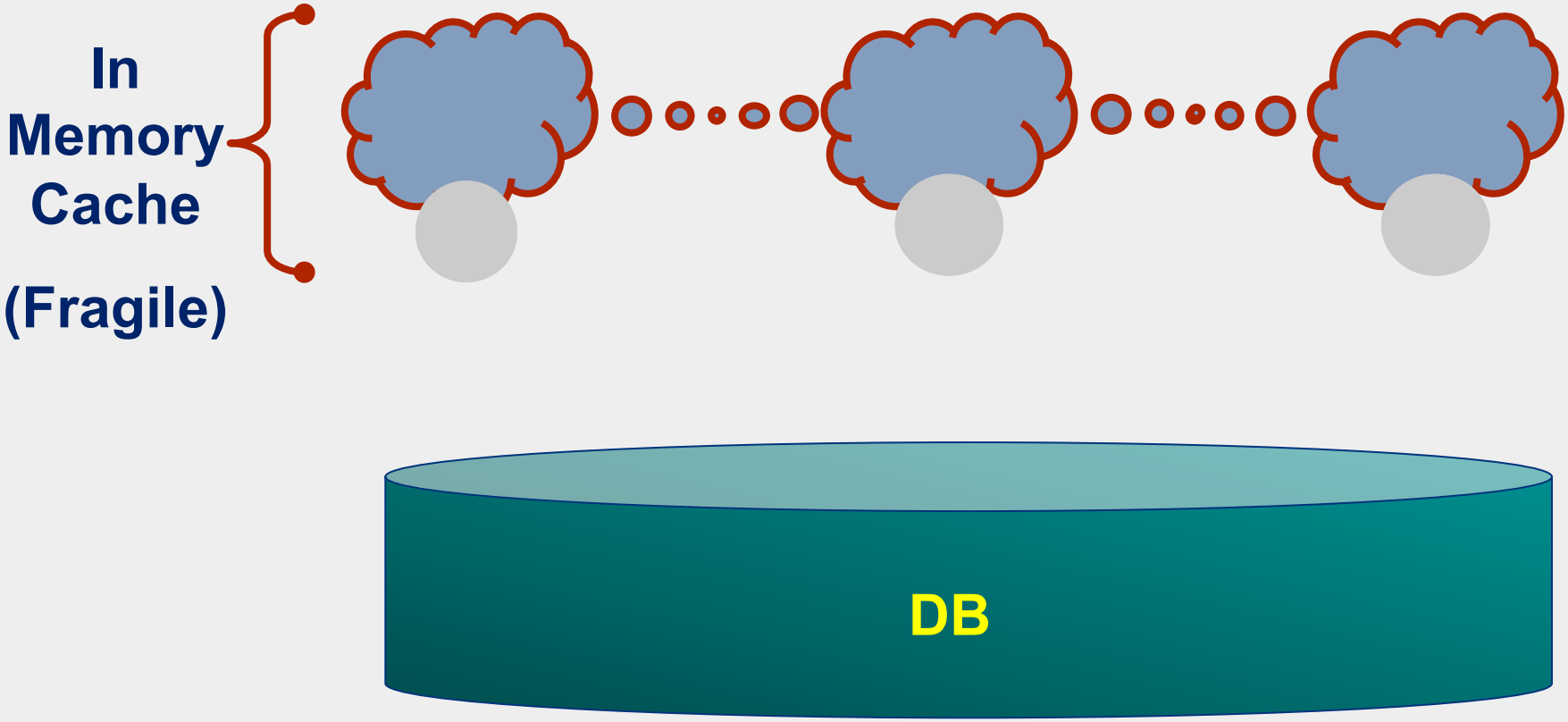


Key Point 2: Coherence gets around the distributed locking problem by partitioning data across multiple machines. This architecture scales.



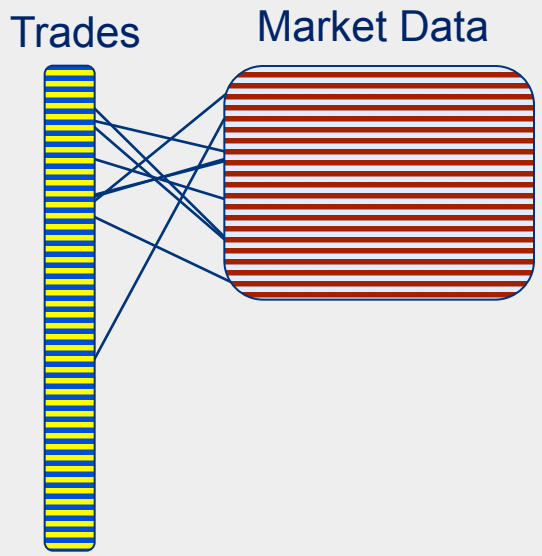
Key Point 3: By working to a simpler contract caches can be much faster and more scalable than databases. They simply have to solve fewer problems.

Key Point 4: Coherence is cleverly optimised for speed but at the price of resilience

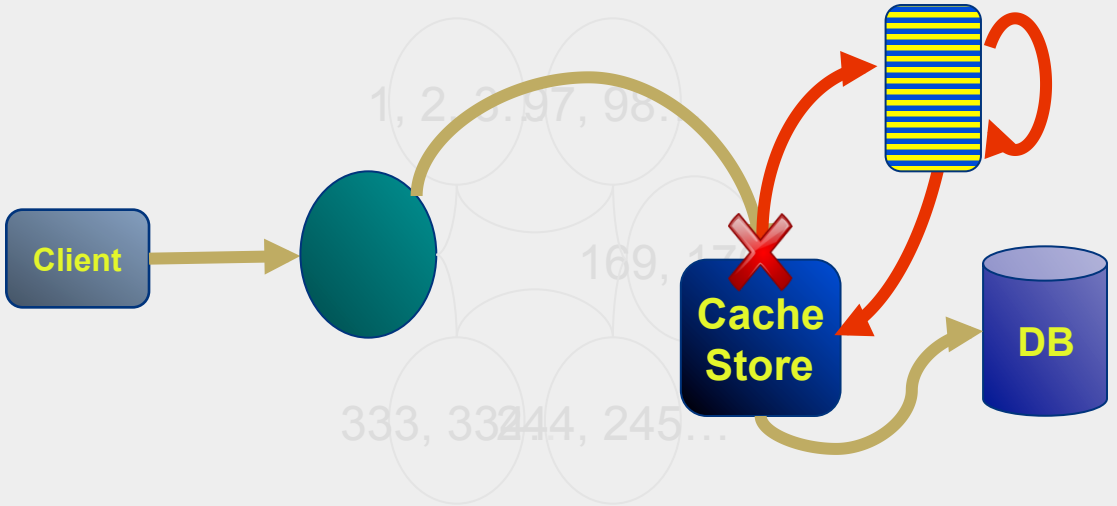


Key Point 5: Advanced functionality facilitates asynchronous distributed processing.

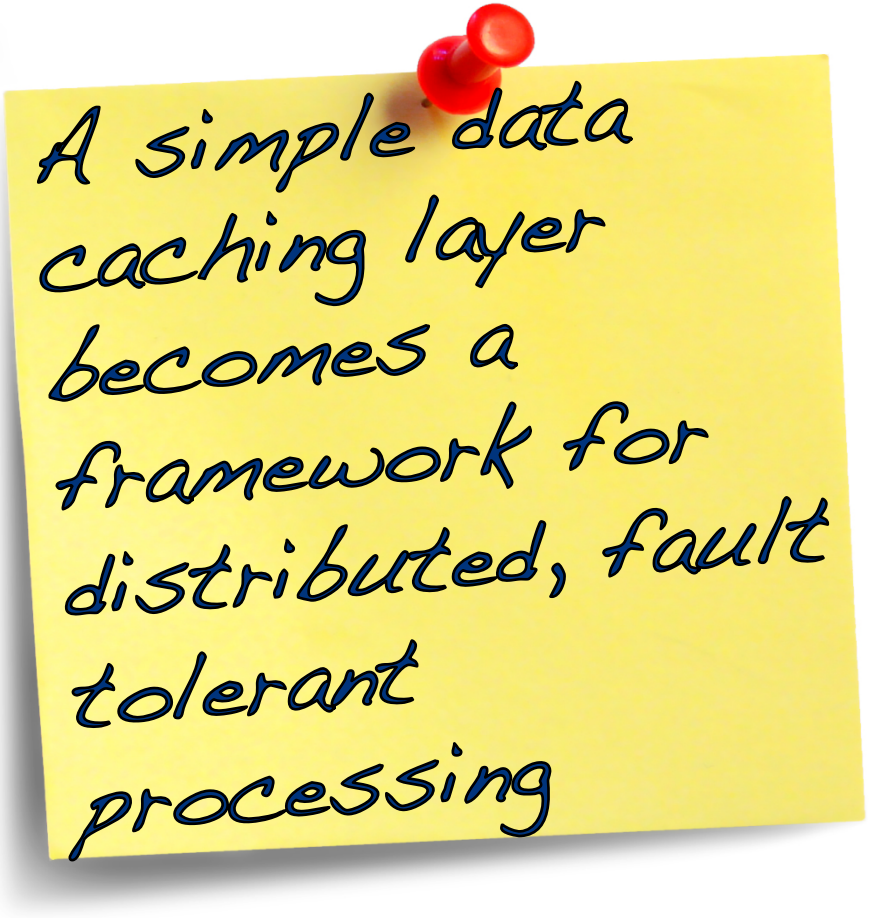
Data affinity



Reliable Processing



So in conclusion…



Summary so far…