Distributed Caching: Essential Lessons

Stuttgart JUG

19 June 2006

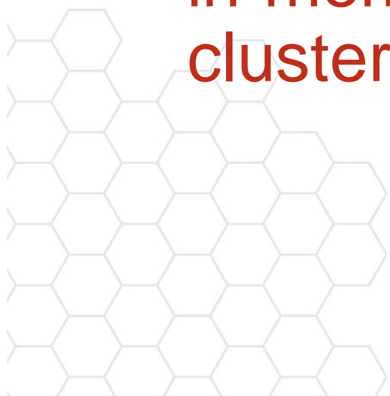
Cameron Purdy

Agenda

- Introduction
- Grid Overview
- Distributed Caching & Data Grid Topologies
- Data Source Integration
- Data Grid Processing
- Essential Lessons
- Customer Use Cases
- Q&A

Speaker's Qualifications

- Cameron Purdy is President of Tangosol, and is a contributor to Java and XML specifications
- Tangosol is the JCache (JSR107) specification lead and a member of the Work Manager (JSR237) expert group.
- Tangosol Coherence is the leading clustered caching and data grid product for Java and J2EE environments. Coherence enables highly scalable in-memory data management and caching for clustered Java applications.

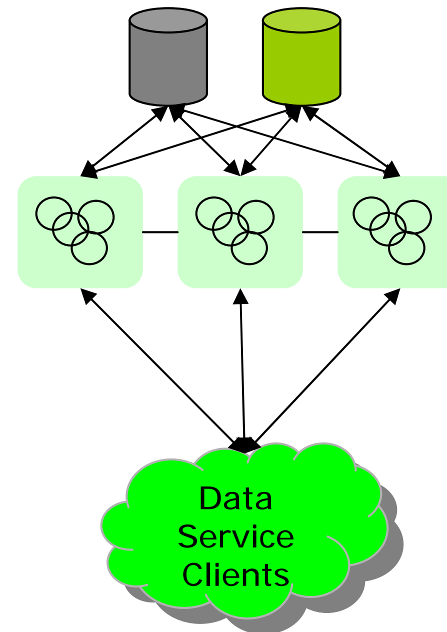


Drivers

- **Data growth** and use has dramatically outpaced existing methods for feeding and managing data between applications and data sources.
- Ensuring **availability**, **reliability**, **scalability** of mission critical applications has evolved from difficult to near impossible as applications have become public-facing, concurrent user load has increased dramatically, and failures are public.
- Pressures on infrastructure and operational **costs** are driving organizations to look for alternatives.

Solution

- **Data as a Service**
- **Horizontal Scale**
  - COTS & Commodity Hardware
  - Generic Switched Network Infrastructure
- **Data Integration is in the Data Service**
  - Not in the database!



Tangosol Coherence

- Provides applications **instantaneous access to data** in memory across the grid achieving unprecedented **application scalability**.
  - Power to handle highest data volumes
  - Reduced latency dependency to back end data source
- Enables **on-demand provisioning** of data enabling greater predictability of cost and capacity.
  - More efficient utilization of infrastructure
- Enables the enterprise to dramatically **improve service levels** by ensuring that application capacity and the data they need are available and responsive 24/7.
- Greatly **reduces grid complexity** by brokering data between data sources and applications dynamically and transparently.
  - Reduced development/deployment/operational resources

Tangosol Coherence: What is it?

- The Data Grid
- The Information Fabric
- The Distributed Shared Memory
- The Single System Image
- The Distributed Data Service
- The Distributed Cache
- The Coherent Clustered Cache

The Basics of Grids and Data Grids

Define Grid

- **There are two distinct meanings**
  - Using an array of servers to create a mainframe-class processing service, e.g. many different and potentially unrelated jobs can be submitted and monitored, and the results come back when they are done
  - Using multiple servers to run a single application to improve its throughput and availability

Define Grid

- **The two meanings do not contradict in reality**
  - The first meaning is actually an example of the second meaning (the provisioning and job management service itself is the application)
  - Applications built to the second meaning are quite often being deployed to grids built with the first meaning

Define Data Grid

- **A data grid combines data management with data processing**
  - In a grid, data management refers to the ability to access and manipulate read/write information across any number of servers
  - Because the amount of processing power is immense in comparison to the amount of network bandwidth, data processing should be localized as much as possible

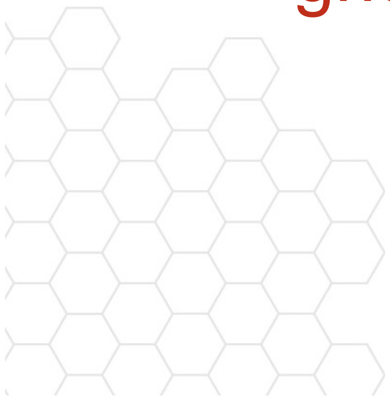
Concepts

- **There are one two things you can move in a distributed environment: State and Behavior.**
  - The distribution of state is often referred to as “replication”, “distributed caching”, etc.
  - The distribution of behavior has traditionally been referred to as RPC, RMI, etc.
  - Data grids combine these two concepts

- **To process data, you can either move the data to where the processing is, or … move the processing to where the data is**
  - Moving the parameters of execution is usually much lighter than moving the required data
  - Often eliminates the overhead of distributed transaction management
  - Supports easy parallelization of work

Concepts

- **You need to be able to move both**
  - Distribution of state allows lots of servers to refer to the same data for their processing
  - Distribution of behavior allows lots of servers to process in parallel
  - Distribution of behavior also allows processing to occur on the server within a grid that has the best *locality of data*



Concepts

- **Locality of data**
  - Most applications spend most of their time waiting for data
  - This is still true in a distributed environment, even if all the data is in-memory in the grid
  - If the data is partitioned into non-overlapping regions, the behavior can be moved to the server that “owns” the data to process

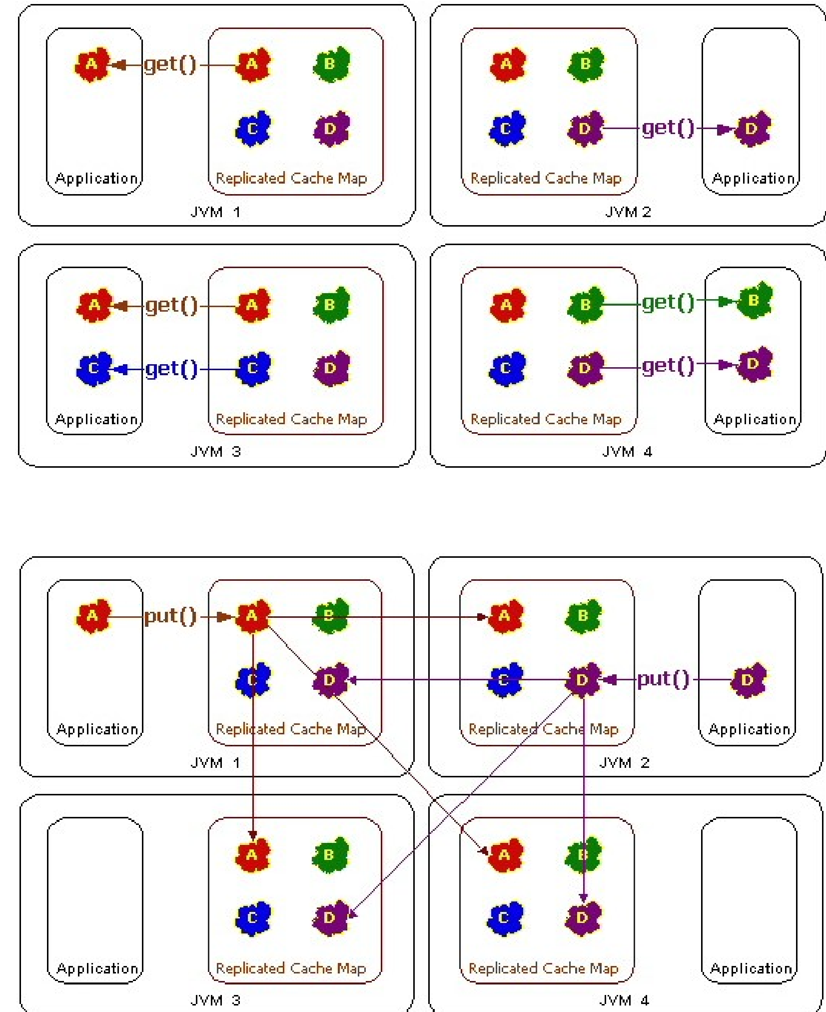
Distributed Cache & Data Grid Topologies

(Or: How to move state.)

TANGOSOL™

Replicated Topology

- Goal: Extreme Performance.
- Solution : Cache Data is Replicated to all members of the cluster.
- Zero Latency Access : Since the data is replicated to each cluster member, it is available for use without any waiting. This provides the highest possible speed for data access. Each member accesses the data from its own memory.
- Limitations
  - Cost Per Update : Updating a replicated cache requires pushing the new version of the data to all other cluster members, which will limit scalability if there are a high frequency of updates per member.
  - Cost Per Entry : The data is replicated to every cluster member, so Java heap space is used on each member, which will impact performance for large caches.



TANGOSOL

Replicated Topology: Summary

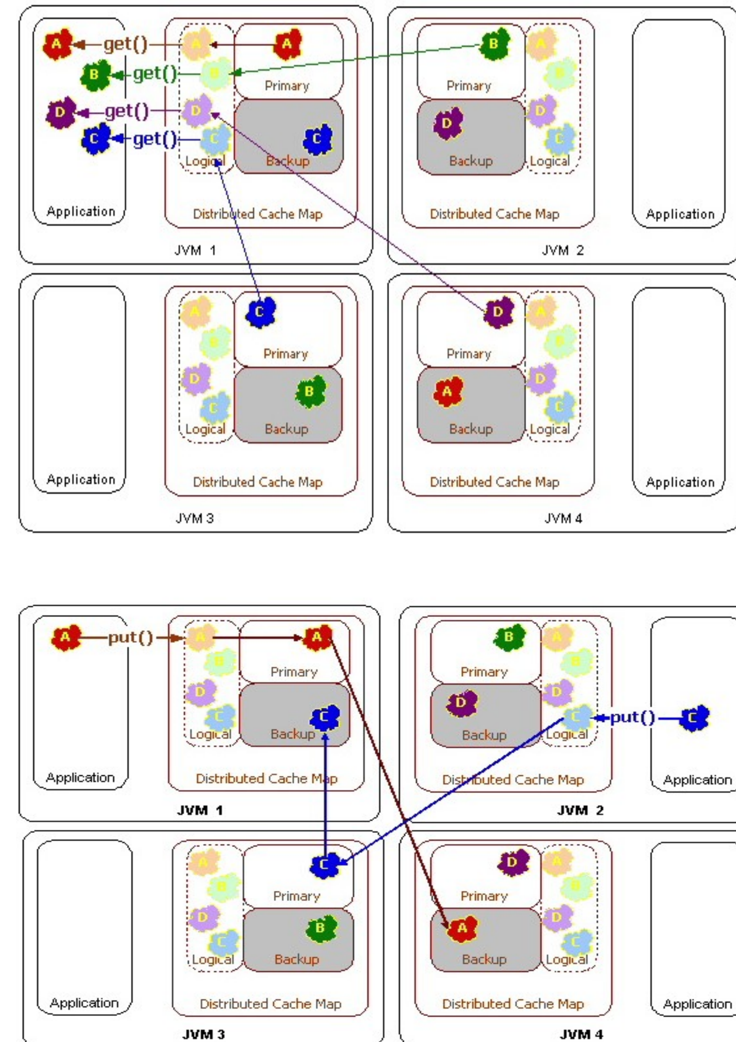
- **Performance**: Very good read performance
- **Scalability**: The scalability of replication is inversely proportional to the number of members, the frequency of updates per member, and the size of the updates
- **Uses**: Small read-intensive caches that benefit from a data “push” model

Partitioned Topology

- **Goal** : Extreme Scalability.
- **Solution** : Transparently partition the Cache Data to distribute the load across all cluster members.
- **Linear Scalability** : By partitioning the data evenly, the per-port throughput (the amount of work being performed by each server) remains constant.
- **Benefits**
  - **Partitioned** : The size of the cache and the processing power available grow linearly with the size of the cluster.
  - **Load-Balanced** : The responsibility for managing the data is automatically load-balanced across the cluster.
  - **Ownership** : Exactly one node in the cluster is responsible for each piece of data in the cache.
  - **Point-To-Point** : The communication for the distributed cache is all point-to-point, enabling linear scalability.

Reads

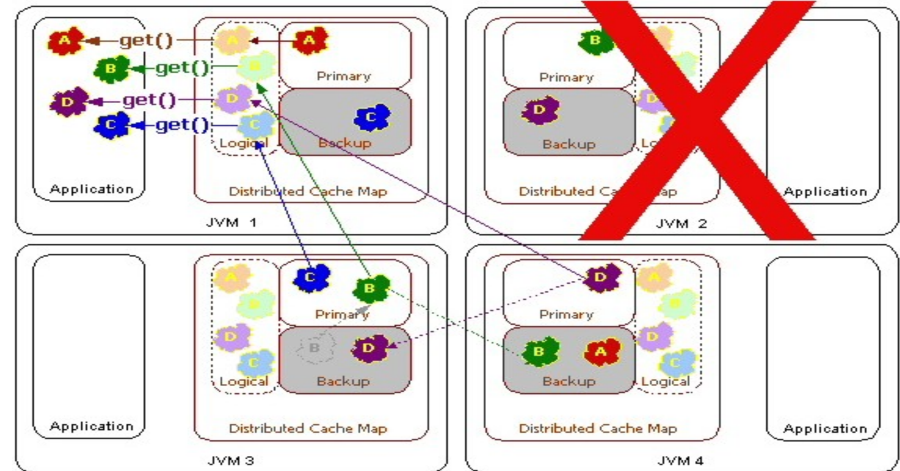
Writes



TANGOSOL

Partitioned Topology: Failover

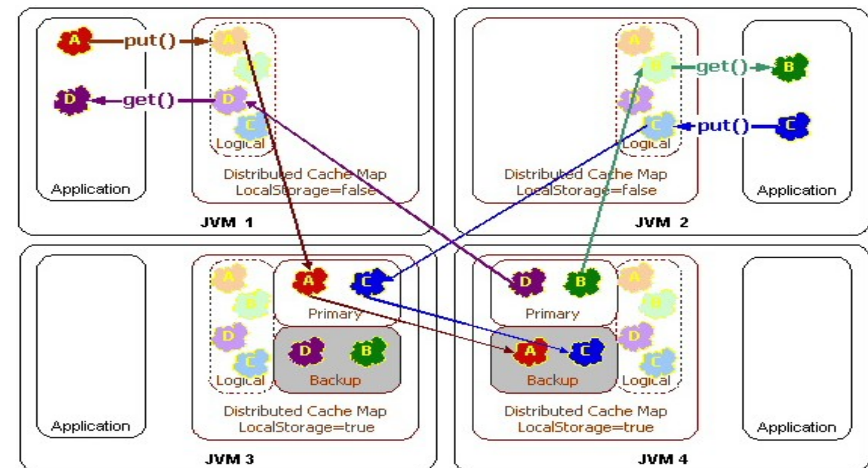
- **Failover** : All cache services must provide lossless failover and failback
  - Configurable level of redundancy
  - Data is explicitly backed up on different physical servers (mesh architecture)
  - There is never a moment when the cluster is not ready for any server to die: No data vulnerabilities, no SPOFs



TANGOSOL

Partitioned Cache Servers

- **Location Transparency**:
  The JCache API and its behavior are the same with a local, replicated or partitioned cache.
- **Local Storage Enabled**:
  Cluster nodes with local storage enabled will provide the cache and backup storage for the partitioned cache. All cluster nodes will have the same exact view of the data, because of Location Transparency.



TANGOSOL

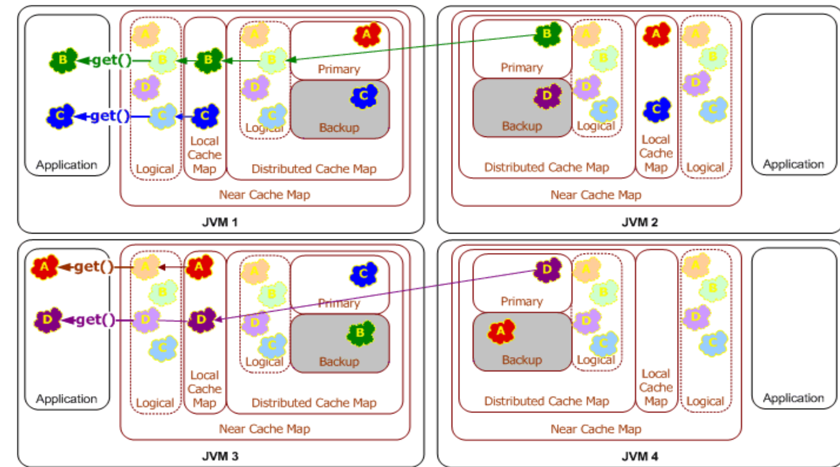
Partitioned Topology: Summary

- **Performance**: Fixed cost.
- **Scalability**: Linear scalability of both cache capacity and throughput as the number of members increases. Designed for scaling on modern switched networks.
- **Uses**: Any size caches, scaling with the size of the cluster or data grid. Both read- and write-intensive use cases. Ability to offload heap usage to other JVMs. Load-balancing. Resilient to server failure.

Near Cache Topology

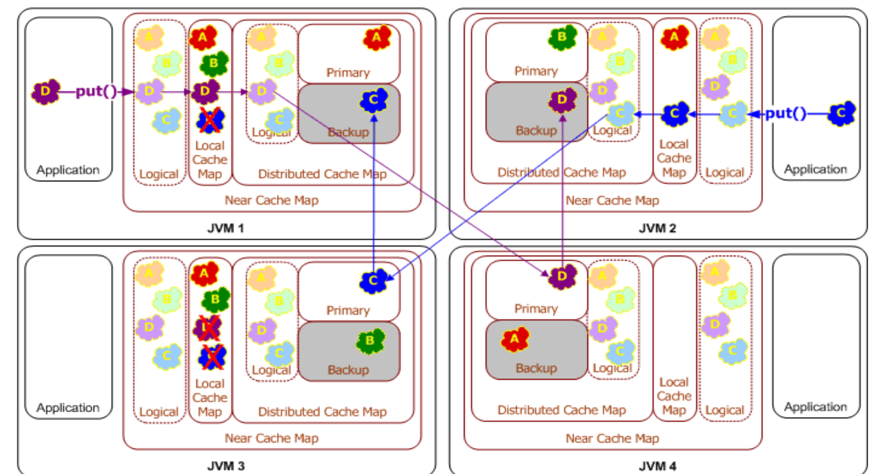
- **Goal**: Extreme Performance. Extreme Scalability.
- **Solution**: Local “L1” In-Memory cache in front of a Clustered “L2” Partitioned Cache.
- **Result**: Zero Latency Access to recently-used and frequently-used data. Scalable cache capacity and cache throughput, with a fixed cost for worst-case.

Reads



Sidebar: Event Capabilities

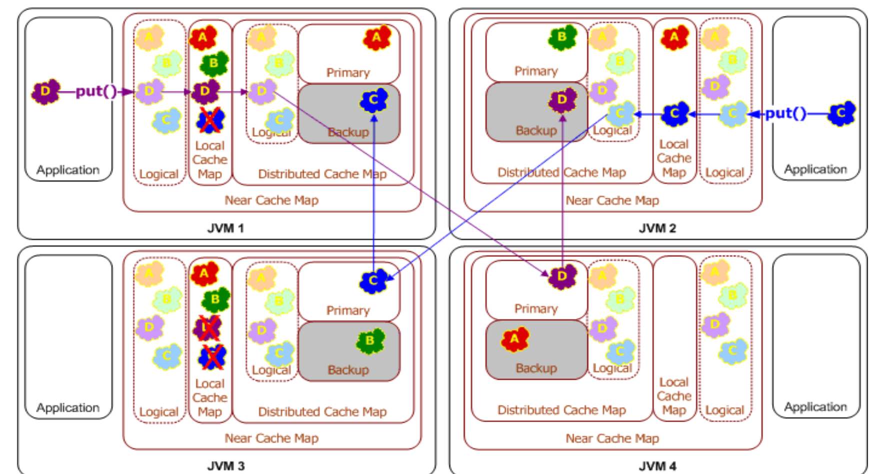
- **Universal:** All caches provide events, regardless of the topology.
- **Clustered:** The events are always delivered efficiently to the interested listeners.
  - Location transparent
- **Powerful Capabilities:**
  - Listen to entire caches, specific identities, or even a query
  - Optionally contain “before” and “after” state
  - Support for both sync and async



TANGOSOL

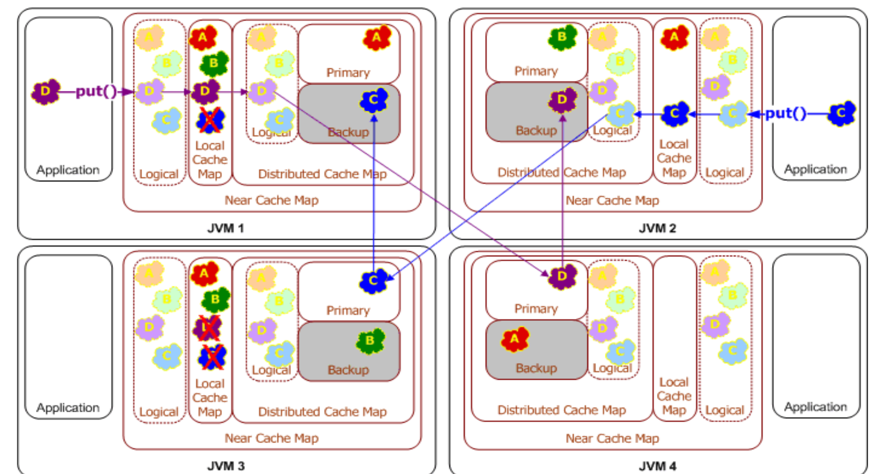
Sidebar: Concurrency Capabilities

- **Explicit:** Caches support explicit pessimistic locking
  - Reliable Clustered Mutex
- **Transactions:** Unit of work management
  - Supports both optimistic and pessimistic transactions
  - Supports all isolation levels from read-committed through serializable
  - Integrates with JTA using the Java EE Connector Architecture



Near Topology: Coherency

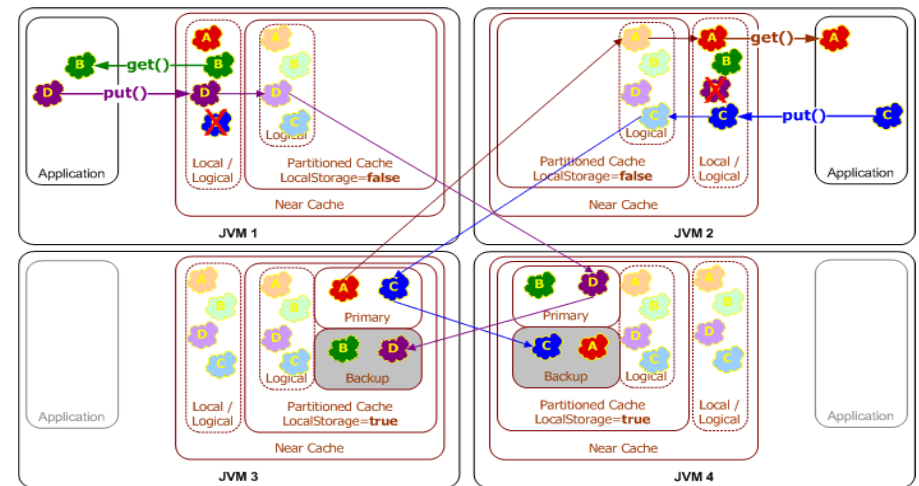
- **Read-Only/Read-Mostly:** Expiry-based Near Caching allows the data to be read until its configured expiry
- **Event-Based Seppuku:** Eviction by event. The Near Cache can automatically listen to all cache events, or only those cache events that apply to the data it has cached locally.



TANGOSOL

Near Topology: Cache Servers

- **Potent Combination**: Combining the benefits of Near Caching with the dedicated Cache Servers can provide “the best of both worlds” for many common use cases.
- **Bulging At The Heap**: This topology is very popular for application server environments that want to cache very large data sets, but do not want to use the application server heap to do so.
- **Balanced**: The application server will use a tunable amount of memory to cache recently- and frequently-used objects in a local cache.
  - Classic space / time trade-off



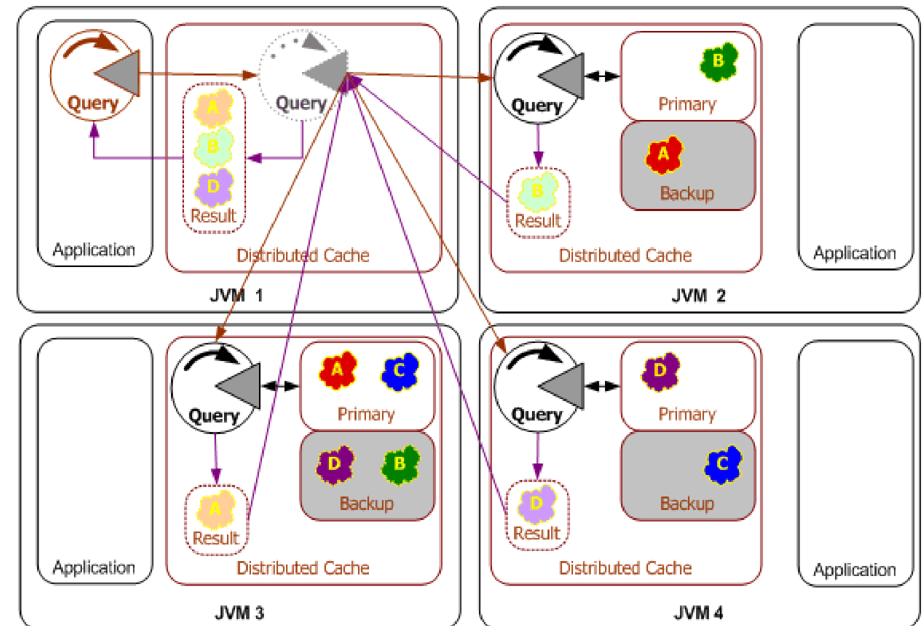
TANGOSOL

Near Topology: Summary

- **Performance**: Zero-latency for common data. Fixed cost for the remainder of the data.
- **Scalability**: Linear scalability of both cache capacity and throughput as the number of members increases. Slightly less with Seppuku.
- **Uses**: Any size caches. Great for read-intensive caches with tight data access patterns. Killer “Cache Server” configuration.

Sidebar: Query Capabilities

- **Parallel Query**: A Partitioned Cache query is performed in parallel across the cluster, using indexing and a iterative Cost Based Optimizer.
  - Customizable predicates
  - Result set paging
- **Continuous Query**: Combines a query with events to provide a local materialized view.
  - The contents are kept up-to-date in real-time.
  - Like a Near Cache that always contains exactly what you need!



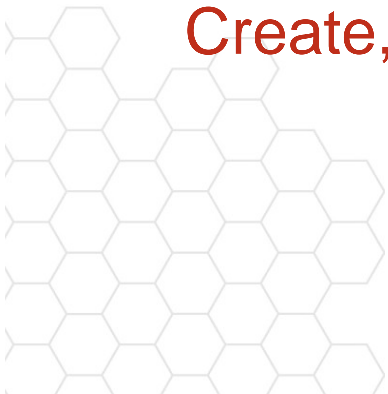
Integration with Data Sources

Cache-Aside Architecture

- Cache-Aside refers to an architecture in which the application developer manages the caching of data from a data source
- Adding cache-aside to an existing application:
  - Check the cache before reading from the data source
  - Put data into the cache after reading from the data source
  - Evict or update the cache when updating the data source

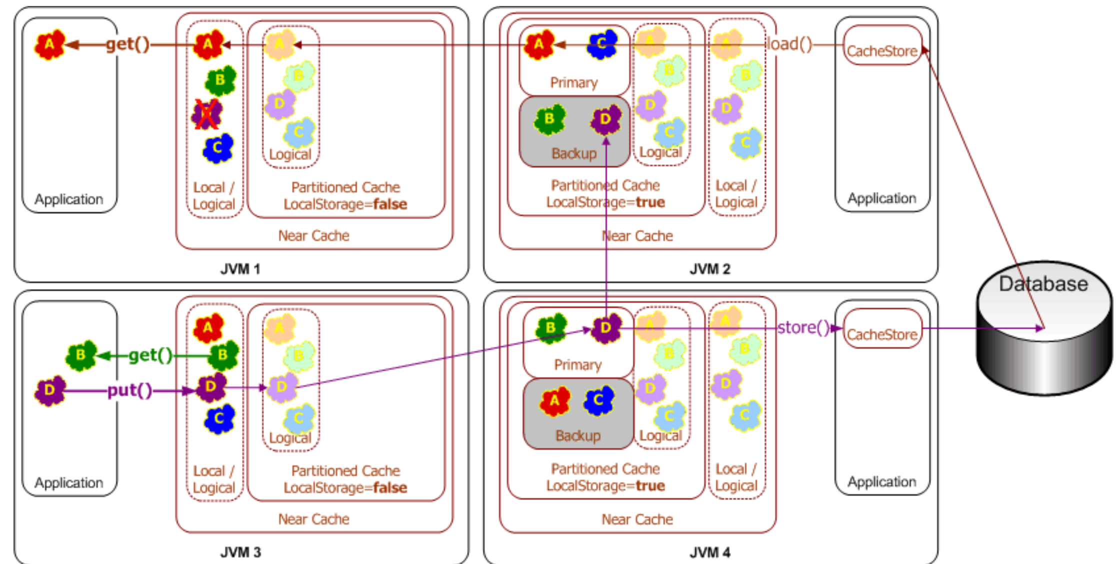
Cache-Through: Architecture

- Cache-Through places the cache between the client of the data source and the data source itself, requiring access to the data source to go through the cache.
- A Cache Loader represents access to a data source. When a cache is asked for data, if it is a cache miss, then any data that it cannot provide it will attempt to load by delegating to the Cache Loader.
- A Cache Store is an extension to Cache Loader that adds the set of operations generally referred to as Create, Read, Update and Delete (CRUD)



Cache-Through: Partitioning

- Cache-Through operations are always managed by the owner of the data within the cluster.
- Concurrent access operations are combined by the owner, greatly reducing database load.
- Since the cache is aware of updates, Write-Through keeps the cache and db in sync.



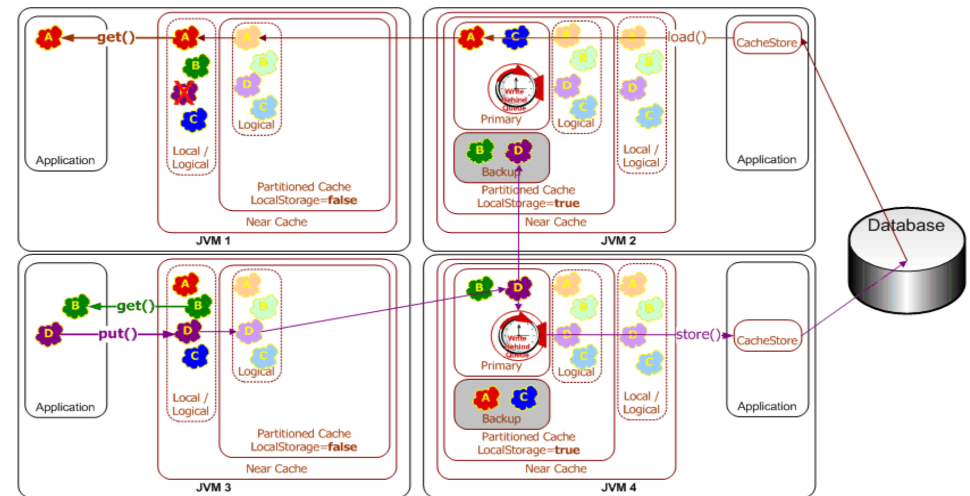
TANGOSOL

Cache-Through: Summary

- **Performance**: Reduces latency for database access by interposing a cache between the application and the data. Database modifications additionally involve a cache update.
- **Scalability**: Channels and combines data accesses. May significantly reduce database load.
- **Uses**: Any time a cache needs to transparently load data from a database.

Write-Behind: Description

- **Write-Behind** accepts cache modifications directly into the cache
- The modifications are then **asynchronously** written through the Cache Store, optionally after a specified delay
- The write-behind data is clustered, making it **resilient to server failure**



TANGOSOL

Write-Behind: Conclusions

- **Performance**: Low-latency for cached reads and writes.
- **Scalability**: The same extreme read/write scalability of the cache, and significantly reduced load on the db
  - Coalesced write-through of multiple modifications to an object
  - Batched write-through of modifications to multiple objects
- **Uses**: When write performance is important, data source load is high, and/or when an application has to be able to continue when the data source is down.
  - Only use when all writes to the data source come through the cache, and db-level auditing is not required

Data Grid Processing

(Or: How to move behavior.)

Coherence Grid Capabilities

- **Node-Based**: Directed execution to specific node(s) or the entire grid:

```
Map map = isvc.query(new CalcAgent(), null);
```

- **Task-Based**: Standard WorkManager API - JSR 236/237
  - The grid becomes one giant thread pool for parallel execution

```
for (int i = 0, c = work.size(); i < c; ++i) {
  mgr.schedule((Work) work.get(i));
}
mgr.waitForAll(work, timeout);
```

- **Data-Centric**: Localizes processing of data
  - Equivalent of a Stored Procedure for a Data Grid
  - Achieves Once-And-Only-Once Processing
  - Supports Entry Processors and Parallel Aggregators

Data Grid Processing

- Compare the cost of moving state versus behavior in a large-scale Data Grid
  - State: lock(id), v=get(id), process, put(id,v), unlock(id)

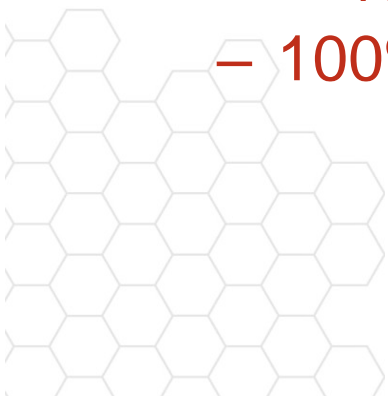
```
map.lock(id, -1);
try {
  Integer I = (Integer) map.get(id);
  int c = (I == null ? 0 : I.intValue());
  map.put(id, new Integer(++c));
  return c;
} finally {
  map.unlock(oKey);
}
```

Data Grid Processing

- Compare the cost of moving state versus behavior in a large-scale Data Grid
  - Behavior: execute(id, process)

```
map.invoke(id, new NumberIncrementor(
   (String) null, new Long(1L), false));
```

  - 72% fewer network hops per process
  - 100% less network-time critical-section



TANGOSOL

Data Grid Processing

- Parallel Processing and Aggregation of Data using Partitioning
  - Analogue to the Parallel Query capability
  - Simple: Just specify a list of identities or a query
  - Linear scale for processing and aggregation throughput

```
map.invoke(setIds, new NumberIncrementor(
  (String) null, new Long(1L), false));
```

```
map.invoke(AlwaysFilter.INSTANCE, new NumberIncrementor(
  (String) null, new Long(1L), false));
```

Data Grid Processing

- Building an EntryProcessor
  - Data Grid calls EntryProcessor.process() locally
  - Target data is available via the Map.Entry interface

```
public class Load
        extends AbstractProcessor
    {
    public Object process(InvocableMap.Entry entry)
        {
        if (!entry.isPresent())
            {
            entry.getValue();
            }
        return null;
        }
    }
```

Data Grid Processing

- Parallel Aggregation
  - Parallel-Aware Aggregators
  - Typically Two Phases
    - Parallel partial aggregation
    - Final aggregation (on the node that kicked it all off)

```
Double avgbal = map.aggregate(setIds, new DoubleAverage(“getBalance”));
```

```
Double avgbal = map.aggregate(filter, new DoubleAverage(“getBalance”));
```

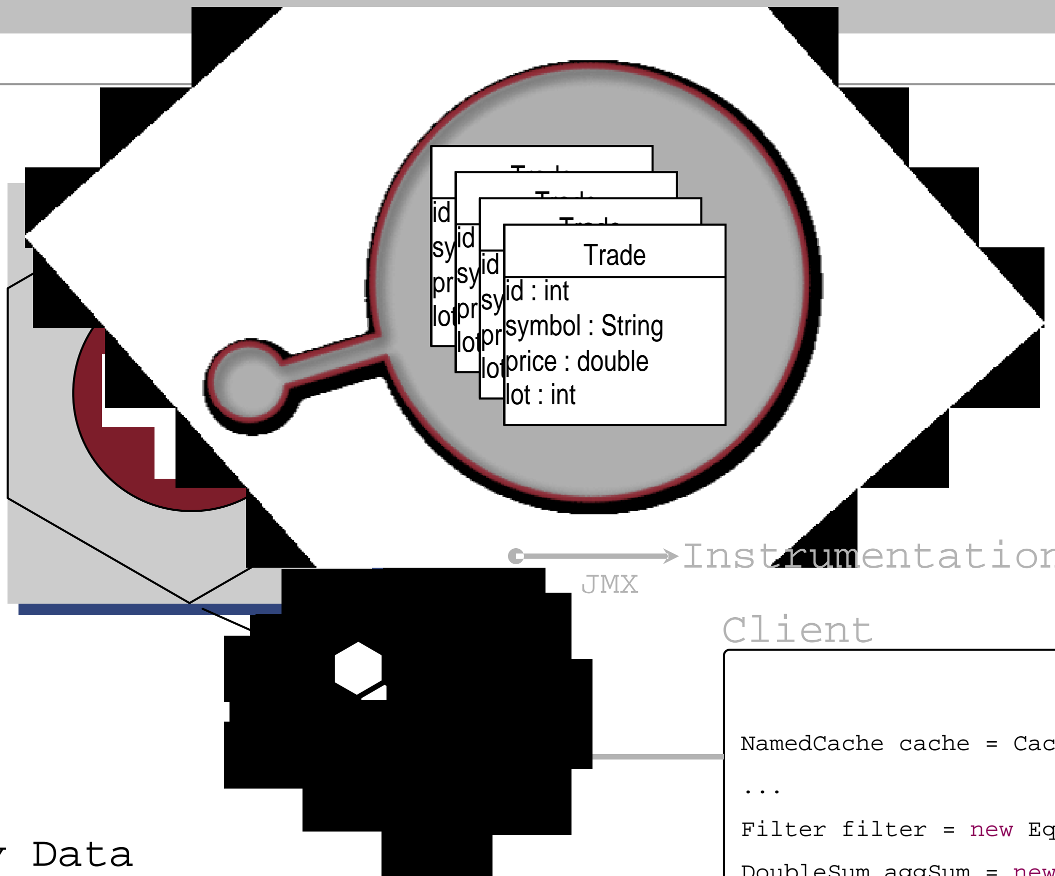
  - Achieves theoretical maximum of Amdahl’s law!
    - (With only one line of code!)

Data Grid Processing

- Biggest Challenges
  - Stable State on Failover during processing
  - Once-And-Only-Once Guarantees
  - Side-Effects
    - Entry Processors can mutate the data in the Data Grid
    - Same is true for Parallel Aggregators
- **Customer Use Cases and Future Focus**
  - Event-Driven Processing
    - Stock Exchange Matching Engines
    - Trading Systems
  - Time Series Aggregations
  - Asynchronous Grid-Wide Event Coupling

An Example of Data Grid: Trading System

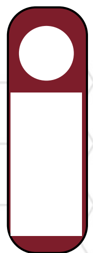
Stock Trade Application



Courtesy of:

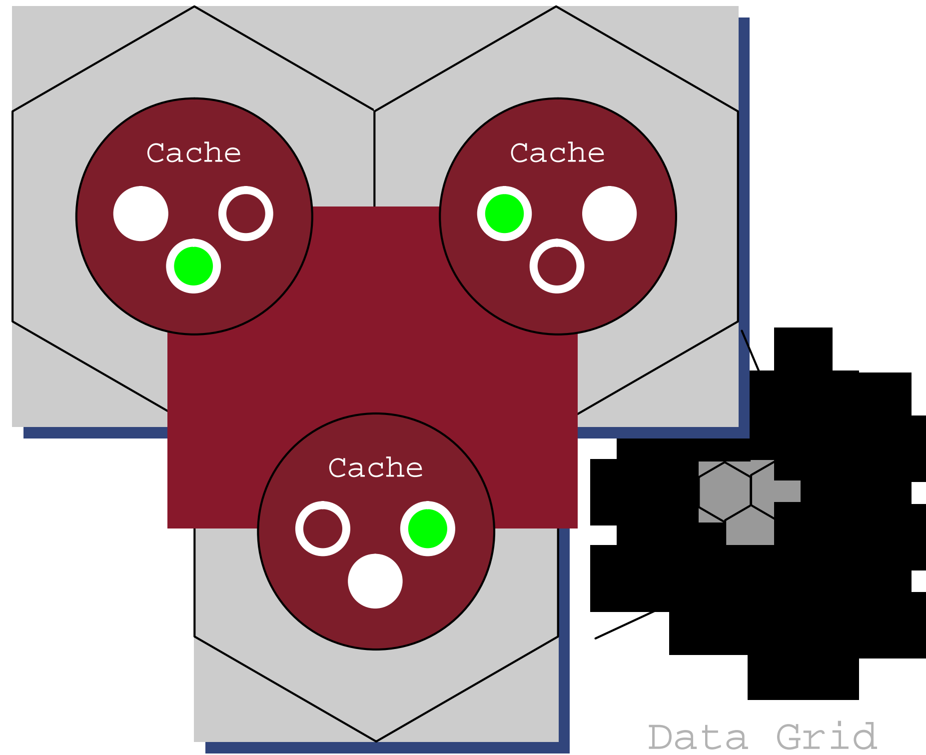
Client

```
NamedCache cache = CacheFactory.getCache("trades");
...
Filter filter = new EqualsFilter("getSymbol", sAggSymbol);
DoubleSum aggSum = new DoubleSum("getPrice");
Double DResult = (Double) cache.aggregate(filter, aggSum);
```



Primary Data
Logical Data
Backup Data

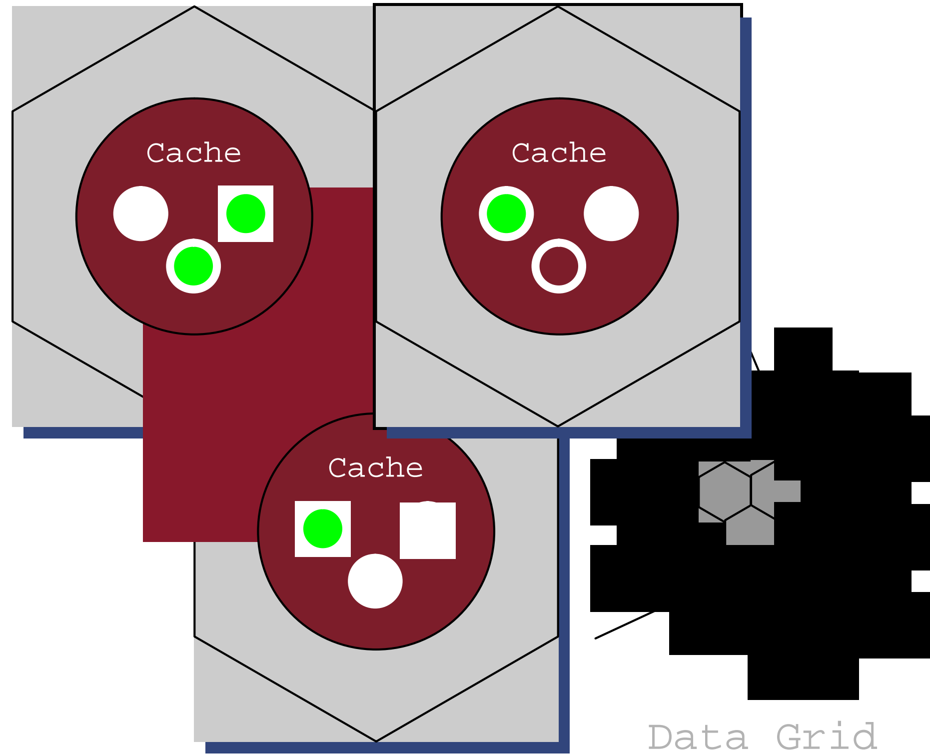
Distributed Cache



Courtesy of:

- Primary Data
- Logical Data
- Backup Data

Failover



Courtesy of:

Primary Data
Logical Data
Backup Data

Lessons for Cache Based Architectures

TANGOSOL™

Lesson 1: Use an MVC Architecture

- **Model/View/Controller (MVC, aka Model2)**
  - Model: Domain-specific representation of the information the application displays and on which it operates
  - View: Renders the model into a form suitable for interaction, typically a user interface element or document
  - Controller: Responds to events – typically are user actions or service requests – and invokes changes on the model

Lesson 1: Use an MVC Architecture

- **Clear delineation of responsibility; for example:**
  - Cache is used in the Model
    - Cache Loader / Store is the DAO
    - Cache contains the application's POJOs / Value Objects
  - View pulls data from the Model
    - Goal is to ensure that all accesses are served from cache
  - Controller affects the Model
    - Modifications via Write-Through or Write-Behind

Lesson 2: Design a Domain Model

- **Domain Model includes two aspects of application modeling**
  - Data Model: Describes the state that the application maintains, both in terms of persistent data (e.g. the “system of record”) and runtime data (sessions, queued events, requests, responses, etc.)
  - Behavioral Model: Describes the various actions that can affect the state of the application. Very similar to the concepts behind SOA, but at a much lower level.

TANGOSOL

Lesson 2: Design a Domain Model

- **Domain Model has value**
  - Allows the Data Model to exist independently of the behavioral model, supporting the separation of a controller from the model in an MVC architecture
  - The behavioral model is the basis for the events that a controller is required to support
  - With an abstract data model that reflects application concerns instead technology concerns, it is much more likely that the resulting data model implementation will be more easily used by the view and the controller

Lesson 3: Specify the Data Access

- **There are multiple ways to access data**
  - ORM (JDO, EJB3, Hibernate, etc.)
  - JCache API (i.e. transparently via a Cache Loader)
  - JDBC (or other direct integration API)
- **Most applications have a "best way"**
  - Large-scale set oriented access usually indicates JDBC
  - Mix of set- and identity-oriented access indicates ORM
  - Identity-oriented access may indicate JCache API

Lesson 3: Specify the Data Access

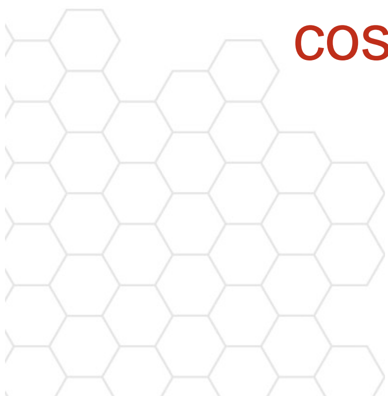
- **Picking the wrong approach is disastrous**
  - RDBMS (JDBC) optimized for set-based queries and operations, including joins and aggregates, but crumbles with heavy row-level access (1+N access pattern, etc.)
  - ORMs can bog down badly on large set-based access
  - JCache API is built around identity-based access, not set-based access

Lesson 3: Specify the Data Access

- **Not always an obvious “best choice”**
  - Some applications have a mix of intensive row-level and large set-level operations, which lend themselves poorly to any single approach
  - Even a well-architected and carefully-designed application will often have a few “exceptions to the rule” that require the specified approach to Data Access to be circumvented
  - It is sometimes necessary to use different approaches for different classes of data within the same application

Lesson 3: Specify the Data Access

- **Optimizations may be available for each**
  - It is often possible to cache JDBC result sets
  - Most ORMs have effective support for pluggable caches, such as Hibernate’s “L2” cache support
  - Some ORMs, such as KODO JDO, have optimizations for set-based operations that can translate some operations directly into optimized SQL that performs the entire operation within the RDBMS
  - Coherence includes parallel query with indexes and cost-based optimizations



Lesson 4: Find the natural granularity

- **Every application has a natural granularity for its data.**
  - Relational data models have a normalized granularity from which tables naturally emerge
  - Optimized JDBC-based applications have a statement execution granularity and a Result Set granularity
  - ORM-based applications and cache-intensive applications often have an OO granularity that mirrors the data model
  - Caches have an identity granularity of access

Lesson 4: Find the natural granularity

- Caches will typically exist for each major class of application object
  - e.g. Accounts, Symbols, Positions, Orders, Executions, etc.
- Each cache will tend to have a natural key
  - e.g. account id, symbol, account id + symbol, order id, etc.
- Application objects tend to be complex
  - Contain “owned” objects, e.g. Purchase Order contains Lines

Lesson 5: Decouple using Identity

- Store, Load, Provide and Manage the Identity of related model objects
- Provide accessors for related model objects by using Identity de-reference (i.e. cache access)
- Read-only models tend to have more lee-way
  - Soft references
  - Transient reference fields

Lesson 5: Decouple using Identity

- Simplifies management of large object graphs
- Enables efficient lazy loading of object graphs
- Works well with …

Lesson 6: Use an Immutable Model

- From the View, the Model should treated as if it is read-only
- From the point of view of the Controller, the model that is shared across threads should be treated as immutable, for example just in case the View is using it on a different thread

Lesson 6: Use an Immutable Model

- Since most applications are not read-only, the Controller does have to modify the data represented by the Model
- When the Controller needs to modify the Model, it can obtain mutable clones of the shared model, and manage them …

Lesson 7: Use Cache Transactions

- … transactionally.
- When the Controller obtains cached values within a transaction, the values are actually clones of the “master” cached values
- The Controller makes its modifications to the Model in a transactionally isolated and consistent manner

Lesson 7: Use Cache Transactions

- For maximum scalability, most transactions should be optimistic. Just as with any optimistic transaction approach, this implies that the application must handle and/or retry transactions whose optimistic checks fail
- Cache Transactions can integrate with the container's Transaction Manager via the JEE Connector Architecture

Lesson 8: Use Queries Wisely

- Cache Queries may be optimized, and they may even be run in parallel across a cluster, but they are probably at least an order of magnitude more expensive than identity-based operations
- If you use queries, make sure to use indexes; for example, the Coherence query optimizer uses multiple indexes on a single query, even if they don't perfectly "cover" the query

Lesson 9: Optimize Serialization

- Objects that are stored in a cache may need to be serialized, and Java's default object serialization is relatively inefficient
- Implementing the Externalizable interface may help slightly
- Serialization using data streams instead of object streams can make a phenomenal impact
  - ExternalizableLite interface
- Serialization performance improvements are up to an order of magnitude, and the reduction in size can be up to 80%

Lesson 9: Optimize Serialization

- Since Java does not have an object-cloning interface, classes that do not have a public clone() method may require serialization and deserialization in order to be cloned
- Since Cache Transactions may need to clone an object to create a copy within a local transaction, optimized serialization can even improve the performance of transactions

Lesson 10: Use Good Identities

- An Identity implementation must provide correct hashCode() and equals() implementations, and cache the hash-code!
- A good toString() implementation helps with debugging (and not just for Identities!)
- If feasible, make your Identity classes immutable
- An Identity must be Serializable, and its serialized form should be stable: Two instances should serialize to the same binary value if and only if equals() returns true
- Java's String, Integer, Long, etc. are perfect

Lesson 11: Cache in the Right Scope

- HTTP Session objects can be used for caching user- or session-specific information; don’t use them as a cache for global information
- Conversely, don’t use a global cache for user-specific caching when the HTTP Session would do just fine

Lesson 12: Never Assume

- **Always verify that it works as expected**
  - We have seen caches in production that literally were not even getting used, and we have seen caches that had not even been configured – or were badly mis-configured
  - Load test and use JMX to monitor what caches exist, how big they are, what their hit rates are, and if the stats are as expected

Case Studies

Case Study: Travel Sites, Web Services

- www.random-travel-site.com
  - If a customer searches on a flight arriving in a city on Wednesday, automatically check hotel availability for Tuesday, Wednesday and Thursday to display with the flight search results
- www.random-hospitality-chain.com
  - Exposes availability as a web service, with date and location as parameters
  - Load was several times (e.g. 3x) higher than expected!

Case Study: Travel Sites, Web Services

- Problem: The hospitality web service doesn't support multiple dates as input, which results in more load than predicted due to some random travel site
- Solution: Caching web service results dramatically reduces load on back end services that provide availability data
- Implementation: Store each web service result in a cache, identified by the request and its parameters. Use an auto-expiry cache and provide the ability for the back-end system to evict data from the cache, ensuring freshness of data.
- Result: Significantly faster web services with significantly less load on expensive back-end systems.

Case Study: MPRPG (Online Gaming)

- www.brand-name-company.com
  - Uses massively parallel role playing games to build brand with young audiences
  - Games are tied tightly to other parts of the business, including various points and reward programs
  - Security is very tight, and all integration with other business units is through services
  - Impressive amounts of concurrent load: Number of players online, amount of player activity

Case Study: MPRPG (Online Gaming)

- Problem: Vertical scale impossible due to socket-based architecture and HA requirements. Online gamers can interact, and the interactions are expected to be occurring in real time.
- Solution: Game state and player state are managed in-memory across the cluster, even though the system-of-record is managed via a service elsewhere within the company.
- Implementation: Use partitioning of areas within the game, and partitioning of data and responsibilities within the cluster. Data services can be invoked and their results can be shared across all servers.
- Result: A successful real-time game engine that scales out on commodity hardware and provides HA.

Case Study: Financial Analysis

- http://some-intranet-site/analysis
  - A large financial services firm uses an internal analysis application to display equities positions, prices changes, trends, and a large number of other information.
  - The information is available from a shared database, but cannot fit into memory.
  - Huge amounts of data from many database tables are required to assemble even a single web page.

Case Study: Financial Analysis

- Problem: Even with static data cached, page times are over 15 seconds! The data set is so large that it cannot fit into memory, and other work occurring against the database renders the entire application unusable at certain times throughout the day.
- Solution: Pre-load the entire data set, partitioned across multiple servers, so page generation does not use the database.
- Implementation: Create a data model that closely reflects the needs of the web application. Create a bulk loading process that loads all data into that model, storing the objects partitioned and load-balanced across all servers.
- Result: Page times 18ms (TTLB), all SPOFs eliminated.

Case Study: Online Broker

- www.random-brokerage.com
  - Application built on a relatively scalable architecture
  - Customer growth has pushed the application past the limits of what can be accomplished with vertical scale
  - User transaction latencies are increasing as a result (requests are backing up, or transactions are queuing)

Case Study: Online Broker

- Problem: Trade volume is high enough that the database is saturated by too many individual transactions attempting to concurrently write data; this causes SLAs to be broken, which results in penalties and lost revenue
- Solution: Batch writes together by using write-behind caching
- Implementation: Data is written to a cluster-durable write-behind cache, which then asynchronously writes batches of data to the underlying database.
- Result: The latency of the cache write is in the low milliseconds, allowing the application to achieve the required SLAs. As a bonus, database load is significantly reduced.

Questions?

Thank You!

- For more information
  - Online Documentation: http://wiki.tangosol.com
  - Discussion Forums: http://forums.tangosol.com
  - Developer Support: support@tangosol.com
  - Charlie Jarvis (EMEA): cjarvis@tangosol.com

+44 1223 451427 (Office)
+44 7775 913298 (Mobile)

Distributed Caching: Essential Lessons

Stuttgart JUG

19 June 2006

Cameron Purdy