DSO Data Structures Guide

About Terracotta Documentation

This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

Release: 3.6
Publish Date: November, 2011 **Documentation Archive »**

Terracotta DSO Data Structures Guide

Introduction

This guide is intended to provide you with the information you need to make informed decisions about which data structures to use in which context, how to maximize their efficiency, and how to avoid certain pitfalls.

Clustered data structures in Terracotta Distributed Shared Objects (DSO) are functionally equivalent to their non-clustered counterparts. However, there are some performance differences between the different data structures in a clustered context that are important to understand when designing a system. Some of the performance differences are inherent to the design of the data structure; for example, the very different synchronization and concurrency characteristics of `Hashtable`, `HashMap`, and `ConcurrentHashMap`. Other performance differences come from the implementation of a data structure that is expected to work in a local context but operates somewhat differently in a clustered context. Still other differences are simply the result of point-in-time implementation details in the Terracotta core engine. Be sure to check this document regularly to see the status of these point-in-time effects.

Unknown macro: {HTMLcomment}

1. 

   Coming Soon

   a. A discussion of cross-JVM latency and concurrency patterns.

4. Latency
5. TODO: describe the effect of cross-JVM latency and how to plan for it – what is "cross-JVM" latency
7. Concurrency
8. TODO: describe the effect of different concurrency patterns and how to plan for them.

Important Concepts for Concurrent Data Structures

A number of important concepts relating to concurrent data structures are discussed in this document. These concepts are defined in the following sections.

Partial Loading

A data structure that can be *partially* loaded can have portions of itself loaded into a client JVM, while other portions remain unloaded until accessed. Partial loading, which is sometimes referred to as "lazy loading," happens on Terracotta clients, not Terracotta server instances.

Logically Managed Objects

One of the main motivations for partial loading comes from the need to efficiently handle *logically managed objects*. Most of the data structures in the `java.util` and `java.util.concurrent` packages that are supported by Terracotta are logically managed, which means they are kept up-to-date across JVMs by replaying the method calls made in another JVM. This is done for performance reasons and, in the case of hash-based structures, for correctness. However, because they aren't managed by keeping track of their physical references to other objects, they do not share the same virtual heap mechanism enjoyed by physically managed objects. (For more information on physically and logically managed objects, see the Concept and Architecture Guide)

Without special support in the Terracotta core, logically managed data structures are loaded in their entirety into a client JVM when they are accessed. In the case of very large data structures, this can lead to a number of application issues:

- The data structure may take a long time to load from the Terracotta server.
- The data structure may take up too much space in the local JVM heap.
- The data structure may exceed the available JVM heap, leading to `OutOfMemoryErrors`.

To solve these problems, Terracotta implements partial loading for some data structures. In the case of certain Map implementations, the entire key set is loaded when the Map is loaded, but the values are not. The values are loaded automatically when the value is accessed. This can have a significant improvement on heap usage for clustered Maps.

Literals and Partial Loading

A data structure with values that are all literals (a set which includes Java primitives) is *not* partially loaded. Only the portion of a data structure with object references for values can be partially loaded. For a data structure with mixed values — both literals and object references — the literals are all loaded while the non-literals are partially loaded. For example, a map with only large numerical or String values is fully loaded. For more information on literals in Terracotta, see the Concept and Architecture Guide.

Keys and Partial Loading

In data structures with key-value pairs, keys are *never* partially loaded. Collections with extremely large key sets or extremely large keys may still have a significant impact on a JVM's memory. Key values should be kept to the smallest size that still serves the application. Key sets that become too large are an indication that the data should be further partitioned.

Specific Data Structures and Partial Loading

Because the partial loading algorithm for each data structure is different, not all data structures are partially loaded. The implication is that data structures that *are* partially loaded have different implementation and performance characteristics under different scenarios. For a list of data structures that can be partially loaded, see Data Structure Characteristics Table.

Synchronization

Synchronization refers to standard Java synchronization, which makes a data structure thread-safe by introducing locking. Without synchronization, data cannot be guaranteed to be correct. Internal synchronization is a necessary characteristic for a data structure to get Terracotta cluster-wide locking, or auto-locking.

Auto-Locking

Auto-locking refers to the locking available on synchronized data structures where Terracotta promotes the local lock to a cluster-wide lock. In Terracotta, auto-locking can be automatic on certain data structures, or configured on other data structures. This guide shows which data structures are automatically auto-locked by Terracotta.

Concurrency

Concurrency, a crucial and integral part of Java, is the ability to perform simultaneous operations on the same data. Data structures have concurrency if they can safely allow concurrency. Terracotta automatically extends the concurrency of certain data structures across JVMs in a cluster.

Latency

Latency is the lag of response to some requested action. In a data structure, latency is the time taken to complete an action on the data. Latency in data structures varies depending on the data-access characteristics of your application.

Locality of Reference

High locality of reference typically implies low latency because required data is available in local memory. Locality of reference is low if the fault rate for data is high.

Data Structure Characteristics Table

The following table shows how a number of the concepts discussed in Important Concepts for Concurrent Data Structures are applied to certain data structures. If a data structure name in the Data Structure column is a link, you can click it to go to a section with more information on that data structure.

Data Structure	Java Concurrent	Java Synchronized	Terracotta Autolocking	Error rendering macro 'html' Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED
array	No	No	N/A	✓
ArrayList	No	No	N/A	No
AtomicInteger	N/A	Yes	Yes	N/A
AtomicLong	N/A	Yes	Yes	N/A
ConcurrentHashMap	Yes	Yes	Yes	✓
ConcurrentStringMap	Yes	Yes	Yes	✓
HashMap	No	No	N/A	✓
HashSet	No	No	N/A	No
Hashtable	No	Yes	TIM	✓
LinkedBlockingQueue	No	Yes	Yes	✓

LinkedHashMap	No	No	N/A	✓
LinkedHashSet	No	No	N/A	No
LinkedList	No	No	N/A	No
POJO	N/A	N/A	N/A	✓
TreeMap	No	No	N/A	No
TreeSet	No	No	N/A	No
Vector	No	Yes	TIM	No

✓ – The characteristic is a Terracotta feature automatically available with this data structure.
YES – The characteristic is part of this data structure or is made available by Terracotta.
TIM – The characteristic is not part of this data structure, but can be added by using the appropriate Terracotta Integration Module (TIM).
NO – The characteristic is not part of or is not automatically available for this data structure.
N/A – Not Applicable.

Selected Data Structures

Additional information is available on the following data types.

Arrays

Package	N/A
Type	List
Available with Java Version	Built-in
Latency Characteristics	
Special Characteristics	

Unlike many other data structures, Java arrays cannot contain mixed types. Arrays contain either literals or object references. An array containing literals will *not* be partially loaded, while an array containing object references will be partially loaded.

ConcurrentHashMap

Package	java.util.concurrent
Type	Map
Available with Java Version	1.5
Latency Characteristics	Slow on full iterations because all segments must be loaded, locked, and traversed. Low-latency if accessing narrow set of values; higher latency if accessing wide set of values.
Special Characteristics	Read access is fully concurrent. Write access is exclusive of read and write access per segment.

ConcurrentHashMap (CHM), unlike a synchronized wrapper around a `java.util.HashMap`, has very good concurrency characteristics that are exploited by Terracotta in a clustered context.

- Unlike synchronized collections which are locked using a monolithic mutex for all operations, CHM uses read and write locks.
- CHM keys are partitioned into separate segments so when exclusive access is required, when for example two threads need to write to the same segment, only the segment is locked instead of the entire data structure.
- Terracotta clustered CHMs have full read concurrency such that all read operations are allowed to occur concurrently.
- Terracotta clustered CHMs have a read locking optimization such that all read lock acquisitions become greedy concurrently on all relevant nodes. This allows read lock acquisition to happen without a network hop to the Terracotta server on every node. For more information on the greedy lock optimization in Terracotta, see the Concept and Architecture Guide.
- Read operations do not have to wait for write operations on other partitions to fully complete.
- While the key set is monolithically loaded, the values are partially loaded.

ConcurrentStringMap

> **Javadocs**
>
> The Javadocs for ConcurrentStringMap can be viewed here.

Package	org.terracotta.modules.concurrent.collections
Type	Map
Available with Java Version	1.5
Latency Characteristics	Low latency due to support for simultaneous fine-grained and global locking.
Special Characteristics	Lock level can be set to Write or Synchronous Write (calling method blocked until acknowledgment is received). Keys must be of String type. Null keys or values are forbidden.

ConcurrentStringMap (CSM) is a Terracotta implementation of ConcurrentMap. CSM has fine-grained locking characteristics, giving it a performance advantage over ConcurrentHashMap in certain environments. Because CSM provides the ability to lock on individual elements, it reduces contention better than CHM, which must lock on an entire segment even for methods accessing a single entry. For methods requiring a global lock, such as size(), CSM provides a global lock simultaneous to element-specific locks.

The CSM locking model can cause a globally-locked key set to become stale briefly. However, data returned by operations on specific keys is always coherent and up-to-date.

Since CSM is not a standard Java data type, you must include the Terracotta Integration Module (TIM) `tim-concurrent-collections` as a library to access the class. You must also add `tim-concurrent-collections` to the Terracotta configuration file. See the Configuration Guide and Reference for more information on how to configure TIMs in Terracotta.

Usage Recommendations

CSM should be considered for clustered applications where lock contention is a problem.

Applications with the potential for spikes in access to shared data stored in a map can use CSM to reduce or even eliminate lock contention. For example, the Terracotta reference application Examinator uses CSM to store exam results. Each ongoing exam is stored in a map element. If a large number of live exams end at the same time, no lock contention occurs because each update operation can lock its element independently. In contrast, CHM segment-locking impairs performance for these types of operations.

CSM can also reduce or eliminate lock hopping in cases where data is partitioned or tends to remain local. For example, if data element <k1, v1> is accessed locally on a JVM, CSM allows locking on the key k1 — no other locks are required. CHM, by comparison, must load and lock the entire map segment containing <k1, v1>. Keys locked on other JVMs must be requested and held until the operation on <k1, v1> is complete.

Hashtable

Package	java.util
Type	Map
Available with Java Version	1.0
Latency Characteristics	
Special Characteristics	

You can install a TIM called tim-collections to add auto-locking to Hashtable. To learn more about TIMs, see the following resources:

- tim-get
- Configuration Guide and Reference

LinkedBlockingQueue

Package	java.util.concurrent
Type	queue
Available with Java Version	1.5
Latency Characteristics	
Special Characteristics	

This queue is partially loaded, synchronized, and autolocked by Terracotta, making it a good choice for shared data that must be in a producer-consumer queue. An important difference between LinkedBlockingQueue and other partially loaded data structures is that items in LinkedBlockingQueue are not cleared once loaded into the JVM.

POJOs

Package	N/A
Type	N/A
Available with Java Version	N/A
Latency Characteristics	
Special Characteristics	

While a POJO is not explicitly a collection, Terracotta will partially load a POJO's object-reference fields. If your code is designed to use POJOs to store, share, and manipulate data, your application can benefit from partial loading. The more your application's POJOs are optimized for partial loading, the greater the benefit.

Literal fields are loaded all at once. See Partial Loading for more information.

Vector

Package	java.util
Type	List
Available with Java Version	1.0
Latency Characteristics	
Special Characteristics	

You can install a TIM called tim-collections to add auto-locking to Vector. To learn more about TIMs, see the following resources:

- tim-get
- Configuration Guide and Reference

Usage Hints

The following sections provide usage hints for related data structures.

array, ArrayList, or LinkedList?

These data structures do not provide concurrency. To cluster with these data structures, your code must provide concurrency by being properly synchronized.

Both array and ArrayList are indexed, ordered collections. However, only array has partial-loading characteristics. While both data structures are fully loaded with respect to their values, ArrayList also loads its top-level objects while array does not.

A top-level object is an object referenced directly by the object reference stored in the data structure.

Data in LinkedList is not indexed, making it a poor choice for fast access. However, if your requirements call for a data structure which can efficiently grow and shrink in size, LinkedList is a good choice.

If an application fully and regularly scans a data structure, partial loading does not add efficiency because the entire data structure must be loaded. ArrayList or LinkedList may be good choices in this case, especially provided good locality of reference to minimize faulting of remote data.

AtomicInteger and AtomicLong

These data structures introduce concurrency when you need to cluster the primitive types integer and long. However, AtomicInteger and AtomicLong can become a bottleneck when they are clustered and accessed too frequently. A classic antipattern is using AtomicInteger as a cluster-wide counter.

CHM, CSM, HashMap, and TreeMap

If a concurrent map is required, and its keys can be strings, always choose CSM for its superior concurrency characteristics. If keys cannot be strings, choose CHM.

If your code is synchronized appropriately (to provide concurrency) and a light-weight map is desired, consider HashMap. While it doesn't automatically provide concurrency, HashMap can be partially loaded. It offers faster response time for gets, and demands less memory.

If your code is synchronized appropriately (to provide concurrency), and a sorted light-weight map is desired, consider TreeMap. However, TreeMap does not support partial loading, and so is efficient only if kept small.

HashSet, LinkedHashSet, and TreeSet

These sets do not provide concurrency or partial loading. If concurrency is a low priority, but preventing duplicate values is advantageous or required in your application, consider these lightweight sets. If your application reads data randomly, HashSet provides an unordered collection which avoids the performance overhead of sorting data. If your application benefits from sorted data, choose LinkedHashSet or TreeSet. LinkedHashSet may provide better performance than TreeSet since its elements are sorted simply on insertion order.

Since these sets cannot be partially loaded, they can have a negative impact on performance if the data stored in a set grows very large.

LinkedHashMap, Hashtable, and Vector

Generally, LinkedHashMap, Hashtable, and Vector are not recommended for clustered data unless code dependencies or other requirements give no choice.

LinkedBlockingQueue

A concurrent, FIFO queue that blocks threads when empty. Use if your application requires a queue that supports concurrency.

Unknown macro: {HTMLcomment}

Factors to Consider When Choosing a Clustered Data Structure

- Reads and writes
  Efficient concurrency is more crucial for write-heavy applications. For a read-heavy application, how data is accessed is important. In general, randomly accessing data across nodes is most efficient if the data is shared and has concurrency. This is because shared, concurrent data structures help minimize faulting and flushing by allowing lock-protected access on different JVMs. With appropriate locking, there is no danger of incoherent data or incorrect and undefined results.

Within a node, an ordered data structure should provide good performance because Terracotta always loads the entire keyset, even when values are partially loaded. with the entire indexed keyset available, random look-ups are faster.

- Concurrency
  If multiple threads must access the data structure, especially for put operations, look for a structure with good concurrency.
- Size of object graphs

Objects that reference large object graphs can quickly use up memory. In this case, applications benefit from data structures that support partial loading.

- Scanning

Partial loading characteristics will not aid an application that regularly scans data structures because the entire data structure must be loaded for scanning.

- How data is accessed – Random?
  LRU eviction should also be a consideration but not supported well by Terracotta as of this writing.