



Repository performance tuning

Jukka Zitting | Senior Developer



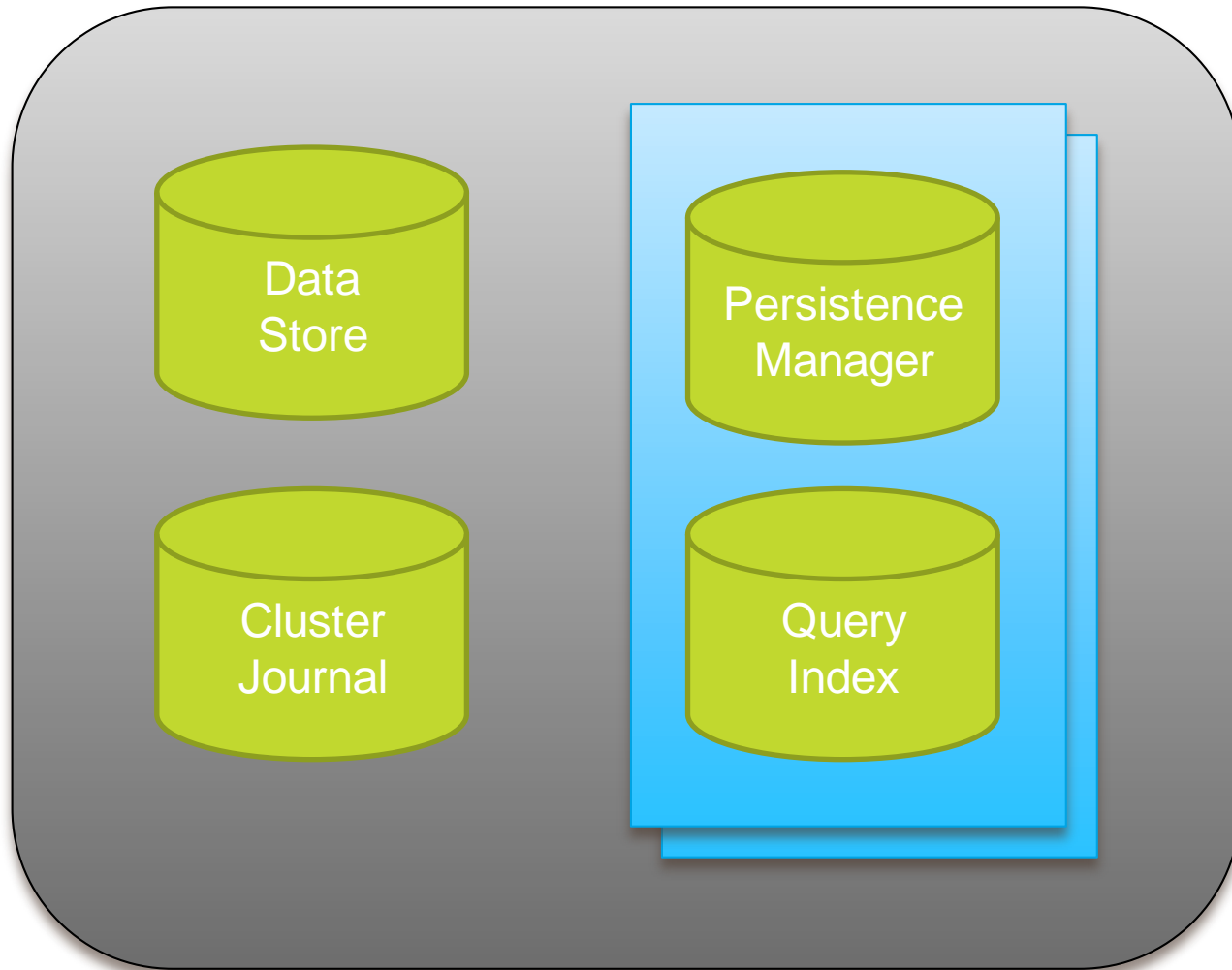
Agenda

- Performance tuning steps
- Repository internals
- Basic content access
- Batch processing
- Clustering
- Query performance
- Full text indexing
- Questions and answers

Performance tuning steps

- Step 1: Identify the symptom
 - Create a test case that consistently measures current performance
 - Define the performance target if current level unacceptable
 - Make sure that the test case and the target performance are really relevant
- Step 2: Identify the cause
 - Main suspects: Hardware, Repository, Application, Client
 - Revise the test case until the problem no longer occurs;
for example: Selenium, JMeter, JUnit, Iometer
- Step 3: Identify/implement possible solutions
 - Change content, configuration, code or upgrade hardware
- Step 4: Verify results
 - If target not reached, iterate the process or revise the goal

Repository internals



Data Store

- Content-addressed storage for large binary properties
 - Arbitrarily sized binary streams
 - Addressed by MD5 hash
 - String properties not included, use UTF-8 to map to binary
- Fast delivery of binary content
 - Read directly from disk
 - Can also be read in ranges
- Improved write throughput
 - Multiple uploads can proceed concurrently (within hardware limits)
 - Cheap copies
- Garbage collection used to reclaim disk space
- Logically shared by the entire cluster



Cluster Journal

- Journal of all persisted changes in the repository
 - Content changes
 - Namespace, nodetype registrations, etc.
- Used to keep all cluster nodes in sync
 - Observation events to all cluster nodes (see `JackrabbitEvent.isExternal`)
 - Search index updates
 - Internal cache invalidation
- Old events need to be discarded eventually
 - No notable performance impact, just extra disk space
 - Keep events for the longest possible time a node can be offline without getting completely recreated
- Logically shared by the entire cluster
 - Writes synchronized over the entire cluster



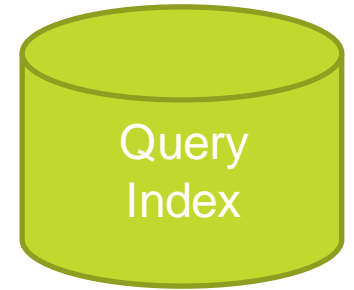
Persistence Manager

- Identifier-addressed storage for nodes and properties
 - Each node has a UUID, even if not mix:referenceable
 - Essentially a key-value store, even when backed by a RDBMS
 - Also keeps track of node references
- Bundles as units of content
 - Bundle = UUID, type, properties, child node references, etc.
 - Only large binaries stored elsewhere in the data store
 - Designed for balanced content hierarchies, avoid too many child nodes
- Atomic updates
 - A save() call persists the entire transient space as a single atomic operation
- One PM per workspace (and one for the shared version store)
 - Logically (often also physically) shared across a cluster



Query Index

- Inverse index based on Apache Lucene
 - Flexible mapping from terms to node identifiers
 - Special handling for the path structure
- Mostly synchronous index updates
 - Long full text extraction tasks handled in background
 - Other cluster nodes will update their indexes at next cluster sync
- Everything indexed by default
 - Indexing configuration for tweaking functionality, performance and disk usage
- One index per workspace (and one for the shared version store)
 - Not shared across a cluster, indexes are local to each cluster node
- See http://wiki.apache.org/jackrabbit/Search#Search_Configuration



Agenda

- Performance tuning steps
- Repository internals
- Basic content access
- Batch processing
- Clustering
- Query performance
- Indexing configuration
- Questions and answers

Basic content access

- Very fast access by path and ID
 - Underlying storage addressed by ID, but path traversal is in any case needed for ACL checks
- Relevant caches:
 - Path to ID map (internal structure, not configurable)
 - Item state caches (automatically balanced, configurable for special cases)
 - Bundle cache (default fairly low, increase for large deployments)
 - Also some PM-specific options (TarPM index, etc.)
- Caches optimized for a reasonably sized active working set
 - typical web access pattern: handful of key resources and a long tail of less frequently accessed content, few writes
- Performance hit especially when updating nodes with lots of child nodes
- FineGrainedISMLocking for concurrent, non-overlapping writes

Example: Bundle cache configuration

```
<!-- In .../repository/worspaces/${wsp.name}/workspace.xml -->
<Workspace ...>

  <PersistenceManager class="...">
    <param name="bundleCacheSize" value="8"/>
  </PersistenceManager>

</Workspace>
```

Batch processing

- Two issues: read and write
- Reading lots of content
 - Tree traversal the best approach, but will flood caches
 - Schedule for off-peak times
 - Add explicit delay (used by the garbage collectors)
 - Use a dedicated cluster node for batch processing
- Writing lots of content (including deleting large subtrees)
 - The entire transient space is kept in memory and committed atomically
 - Split the operation to smaller pieces
 - Save after every ~1k nodes
 - Leverage the data store if possible

Clustering

- Good for horizontally scaling reads
 - Practically zero overhead on read access
- Not so good for heavy concurrent writes
 - Exclusive lock over the whole cluster
 - Direct all writes to a single master node
 - Leverage the data store
- Note the cluster sync interval for query consistency, etc.
 - `Session.refresh()` can be used to force a cluster sync

Query performance

- What's really fast?
 - Constraints on properties, node types, full text
 - Typically $O(n)$ where n is the number of results, vs. the total number of nodes
- What's pretty fast?
 - Path constraints
- What needs some planning?
 - Constraints on the child axis
 - Sorting, limit/offset
 - Joins
- What's not yet available?
 - Aggregate queries (COUNT, SUM, DISTINCT, etc.)
 - Faceting

```
SELECT a.* FROM [nt:unstructured] AS a JOIN [nt:unstructured] AS b
  <PersistenceManager class="...">
    <param name="bundleCacheSize" value="8"/>
  </PersistenceManager>

</Workspace>
```

Indexing configuration

- Default configuration
 - Index all non-binary properties
 - Index binary jcr:data properties (think nt:file/nt:resource)
 - Full text extraction support for all major document formats
 - Full text extraction from images, packages, etc. is explicitly disabled
 - CQ5 / WEM comes with default aggregate indexing rules for cq:Pages, etc.
- Why change the configuration?
 - Reduce the index size (by default almost as large as the PM)
 - Enable features like aggregate indexes
 - Assign boost values for selected properties to improve search result relevance

Indexing configuration

- How to change the configuration?
 - indexing_configuration.xml file in the workspace directory
 - Referenced by the indexingConfiguration option in the workspace.xml file
 - See <http://wiki.apache.org/jackrabbit/IndexingConfiguration>
- Example:

```
<?xml version="1.0"?>
<!DOCTYPE configuration SYSTEM
  "http://jackrabbit.apache.org/dtd/indexing-configuration-1.0.dtd">
<configuration xmlns:jcr="http://www.jcp.org/jcr/1.0"
  xmlns:nt="http://www.jcp.org/jcr/nt/1.0">
  <aggregate primaryType="nt:file">
    <include>jcr:content</include>
  </aggregate>
</configuration>
```

Question and Answers



Adobe